2013-03

SPECIFICATION, VALIDATION AND VERIFICATION OF MOBILE APPLICATION BEHAVIOR

Bonine, Christopher B.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SPECIFICATION, VALIDATION AND VERIFICATION OF MOBILE APPLICATION BEHAVIOR

by

Christopher B. Bonine

March 2013

Thesis Advisor: Man-Tak Shing
Thesis Co-Advisor: Thomas Otani

Approved for public released; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2013	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE SPECIFICATION, VALIDATION AND VERIFICATION OF MOBILE APPLICATION BEHAVIOR		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Christopher B. Bonine			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Acquisition Research Program, Naval Postgraduate School		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public released; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

Mobile devices have, in many ways, replaced traditional desktops in usability, usefulness, and availability. Improvements to computational power, battery life, device capabilities, and user experience will continue to drive people to stop using desktops and solely use mobile devices. Applications are vital to maximize usefulness of these devices. Development of these applications proceeds with a rapidity that surpasses the development pace of the devices themselves. Current methods are inadequate when attempting to verify and validate the behavior of the applications to ensure they perform correctly as the customer expect and correctly with respect to the software specifications. The current V&V methods are limited to environments that do not reflect the typical operational environment for mobile devices. These methods lead to false beliefs that the results of V&V tests prove correctness of the software, when they are only proving that the software works in a non-mobile environment. To solve this problem, we propose that application log files be used to capture the execution behavior while operating in their typical environment. The log file along with customer requirements, represented formally as statechart assertions, will provide a mechanism to conduct automated V&V on the behavior of the application while it was operating in its planned, mobile environment.

14. SUBJECT TERMS Verification and Validation; Mobile Devices; Mobile Applications			**15. NUMBER OF PAGES** 98
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public released; distribution is unlimited

SPECIFICATION, VALIDATION AND VERIFICATION OF MOBILE APPLICATION BEHAVIOR

Christopher B. Bonine
Lieutenant, United States Navy
B.S. Southern Polytechnic State University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2013

Author: Christopher B. Bonine

Approved by: Man-Tak Shing
Thesis Advisor

Thomas Otani
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Mobile devices have, in many ways, replaced traditional desktops in usability, usefulness, and availability. Improvements to computational power, battery life, device capabilities, and user experience will continue to drive people to stop using desktops and solely use mobile devices. Applications are vital to maximize usefulness of these devices. Development of these applications proceeds with a rapidity that surpasses the development pace of the devices themselves.

Current methods are inadequate when attempting to verify and validate the behavior of the applications to ensure they perform correctly as the customer expect and correctly with respect to the software specifications. The current V&V methods are limited to environments that do not reflect the typical operational environment for mobile devices. These methods lead to false beliefs that the results of V&V tests prove correctness of the software, when they are only proving that the software works in a non-mobile environment.

To solve this problem, we propose that application log files be used to capture the execution behavior while operating in their typical environment. The log file along with customer requirements, represented formally as statechart assertions, will provide a mechanism to conduct automated V&V on the behavior of the application while it was operating in its planned, mobile environment.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BYOD	Bring-Your-Own-Device
CIO	Chief Information Officer
CMOS	Complementary Metal-Oxide-Semiconductor
DISA	Defense Information Systems Agency
DoD	Department of Defense
EM	Electro-Magnetic
FM	Formal Methods
FSM	Finite State Machine
R&D	Research and Development
V&V	Verification and Validation
U.S.	United States
U.S.S.R	Union of Soviet Socialist Republics

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wonderful wife, Ashley, and my daughters, Isabelle and Evelyn, for their patience and support. Their love allowed me to believe in myself during this arduous process. I love you, Noodles.

This thesis would have not been a success without the time and effort provided by Professor Man-Tak Shing. His experience, and professionalism influenced my decision when I selected a thesis advisor. His availability for questions at any time for any thing allowed me to produce a better product.

Finally, I would like to extend my undying appreciation to Andie and Becky McDaniel, who provided proofreading with a weathered eye.

THIS PAGE INTENTIONALLY LEFT BLANK

I. PROBLEM CONTEXT

Prior to the turn of the century, the Department of Defense (DoD) and to a larger extent the military has been in an interesting position. If DoD wanted a new capability, it was required to develop it, from start to finish. This was valuable since the DoD could dictate how exactly the product would work, and could more or less control the quality of the resulting product. The disadvantage was the end cost to the organization requesting the new capability. Developing a new technology costs significantly more than to improve existing products. This was of great benefit to the non-government sector because after a new defense technology was available long enough that its secrecy was no longer required; it could quickly be integrated into civilian life. This is termed "spin-off" (Stowsky, 2005). There are many examples, such as GPS, the Internet, satellite communications, and supercomputers (Alic, 1992).

In the 1960s, Defense spending increased significantly because of the space and arms race with the former U.S.S.R. As a result, contracts became "vastly more expensive and complicated" (Office of Management and Budget, n.d.). DoD Research and Development (R&D) spending doubled from $2.1 billion in 1958 to over $4.9 billion and would reach over $8 billion by 1968. (Office of Management and Budget, n.d.) A similar increase occurred between 1979 and 1989. These increases in spending were indicative of the Cold War era. Since the United States could not compete with the U.S.S.R. in military numbers, the U.S. needed to be able to dominate the U.S.S.R technologically. This has required ever-increasing spending to continually develop new technology. Combining the high costs of R&D with the cost of paying other private companies to develop the technology has caused the costs to skyrocket. As development costs for replacement systems go up, older systems are required to stay online longer. As systems get older, the costs to maintain them increase, reducing the amount of money that can be spent on developing new technology

As the rate at which the DoD could afford to develop new technology slowed, and the increasing demand that Americans have to new technology, the corporate sector began to develop separately from the DoD. These private companies are better suited to

developing new technology because they can successfully market the products and make money for further development. This leads to faster development cycles, as well as more research into usability: the faster a product reaches the market and better people like it, the more profit that is made. This is the complete opposite from typical DoD-sponsored development, in which the main goal is to reduce cost.

This development cycle leads to the private sector creating products that are closer to state-of-the-art: both quicker and eventually cheaper when mass production begins. While the private sector cannot produce everything that the DoD would need, the products that are successfully marketed could be utilized to create a better "bang to buck ratio." The term "spin-on" refers to the times that private corporations develop a tool or technology that the DoD acquires for its purpose (Stowsky, 2005). Some examples of spin-on that have occurred are CMOS semiconductors, and very-high-speed integrated circuits (Alic, 1992). A current example of spin-on is the movement for adoption of smart phones and smart tablets for use by the DoD.

Companies such as Motorola, Apple, Samsung, and Nokia are producing multiple hardware platforms for mobile devices using operating systems such as iOS, Android, and SymbianOS. New mobile devices and updated operating systems are coming out at a furious rate and at very low cost to consumers. These low-cost devices potentially can be of great benefit to the DoD, which by nature operates in a very mobile environment, both at home and abroad. According to Teresa M. Takai, the CIO for the Office of the DoD, in the DoD Mobile Device Strategy Memorandum, "Department of Defense forces have been and continue to be increasingly mobile." The DoD CIO stated that even though phones are the latest technology in the market, it "is not about embracing the newest technology, it is about keeping the DoD-workforce relevant in an era when information and cyberspace play a critical role in mission success." This may be a primary reason for the use of mobile devices in the DoD: use of commercial mobile devices adds additional benefits.

A. BENEFITS OF MOBILE APPLICATIONS

There are several benefits to using mobile devices developed in the private sector.

- Eliminate Development Cost: Others have already paid for the cost of developing the device.
- Increase Capability-to-Cost ratio: The cost of purchasing these highly capable devices is much lower than to purchase devices developed specifically for the DoD. This is partly due to the lack of development cost of the device.
- Reduce Cost of Training: Nielson Company predicts that by Mar 2012, 50% of Americans will own smartphones. Since the DoD is a subset of the general public, we could say that 50% of DoD workers own a smart phone. This means that the cost of training people on use of the devices over proprietary devices should be much less.
- Leverage existing commercial applications: The DoD could utilize existing applications rather than spend money unnecessarily to produce the same product.

While the DoD may be able to take advantage of the fast development rate and low cost of the hardware, it will be more restricted in operating systems it may use. The U.S. Army has already approved Android 2.2 for use, though it has not been applied to any operational environment outside of testing. Since the DoD has security concerns, the operating system on a device must be thoroughly tested, meaning adoption of a new OS will be slowed. This should not impede the DoD from taking advantage of faster hardware on newer devices using previously approved operating systems.

The final pieces of the puzzle are the applications that will be developed for use by the DoD. These applications give the mobile devices their true capabilities. As stated in the DoD Mobile Device Strategy Memorandum, desired capabilities include:

- Real-time mapping while in the field
- Data overlay capabilities on the real-time maps
- Identification of Friendly Forces while in the field
- Identification and ordering of parts through pictures
- Access to more information (e-mail, intelligence, collaboration)

While these cover many important areas, there will always be new capabilities that will be needed and could be implemented by creation of new applications or

evolving existing ones. If hardware and software are designed and executed well, new hardware would not be needed to add a new capability, saving very limited resources.

The DoD Mobile Device Strategy Memorandum states that there are three "critical" areas to mobility. They are the wireless infrastructure, the mobile device, and the mobile applications. The memorandum articulates the DoD goals regarding the first and the last of these critical areas. The goals for the DoD are:

- Advance and evolve the DoD Information Enterprise infrastructure to support mobile devices
- Institute mobile device policies and standards
- Promote the development and use of DoD mobile and web-enabled applications

Goal 1 revolves around construction of the DoD networks to be able to accommodate mobile devices and applications. Also included is how to manage the EM spectrum since it is a finite resource that will be shared by almost 1.5 million military personnel and over 700,000 civilians who work for the DoD. The additional problem of personnel overseas affecting less capable wireless networks is a complication as well. Creating the security architecture that is able to protect the networks in this constantly-changing and evolving arena also falls under here. There is extensive research ongoing to solve all these problems and will not be discussed further here.

Goal 2 covers use of mobile devices will be used both for official and unofficial purposes. With a user base that could approach or exceed 2 million, a well-defined, specific set of policies that covers a broad set of topics is vital. Areas include security consistency, and interoperability between agencies for information sharing. The structuring of the approval process plays a key role in the overall success of the program. If devices or applications cannot get approved in a timely manner, then the capabilities of the devices will cease to evolve. To manage mobile devices and applications for the DoD, DISA has established an office with that purpose (Kenyon, 2012). This newly established office would ensure "continuous and secure mobile device operations and maintenance in a cost-efficient manner" (Department of Defense, 2012). The office will become responsible for all areas once mobile devices become normal. While not explicitly stated, DISA should provide a "backstop" to ensure that multiple applications with the same

purpose are not developed to prevent unnecessary spending. Education and training are also listed under goal 2. Beyond daily usage of the device, training will cover proper usage as well as how to keep the device software updated. New policies and standards are needed to properly address and take advantage of the DoD workforce's desire to bring-your-own-device (BYOD) to work. Allowing employees to use their personal devices on DoD enterprise networks allows for further cost savings, but introduces new issues such as proper use, security, what authority DoD has in managing the use of the personal device, and if the personal usage is auditable by the DoD since it shares the same hardware with government property.

Goal 3 focuses on the development of applications for the devices. As stated in the DoD Mobile Device Strategy Memorandum, the appeal of mobile apps is the "low-cost, often faster development and delivery of simple but useful function to the warfighter…" The apps are the reason that mobile devices have become so popular in the consumer market, and also drive the reason that the DoD is moving to the use of such devices. The authorized use of the devices will drive what applications can and will be developed. If personnel were only allowed to use the device at work, then only work-related apps would be developed. If the device can be used outside of work for personal life, then a wide variety of applications would be developed. While the latter situations would allow more applications, a set of criteria would need to be established to ensure that the applications on the phone adhered to the organization's standards and security requirements.

Also under Goal 3 is the need to develop a method of distribution of approved applications. DISA is moving forward with an app storefront similar to Apple and Google. The intended result of such a design is to promote "the discoverability and reuse of DoD-approved mobile apps" (Department of Defense, 2012) to reduce cost of both development and life cycle maintenance of overlapping or duplicate applications.

B. MOBILE APPLICATION VERIFICATION AND VALIDATION

In searching a storefront of available apps, finding an application that suits the needs of the user can be daunting. An application is designed for a specific purpose may

or may not suit other purposes. For a user in an office environment, a malfunction of an app may cause an inconvenience, but due to the unusual nature of the DoD, even app issues in an office environment can cause life-threatening situations abroad. The danger to users by a malfunction of an app while deployed in a warzone greatly increases the need for verifying that an app performs as needed and as expected.

This thesis proposes a solution for quickly being able to evaluate an application for correctness when compared to a set of requirements. By requiring that apps created for the DoD produce log files of the application behaviors, a verification and validation process using state charts can be performed. A set of state charts will "assert" the correct operation of the application based on the user requirements. The application will produce a log file that represents its behavior that can then be converted into a set of test cases. These test cases will then be run on the set of state charts and will show if the application is suitable for the user requirements.

This technique can be used to verify that the software does correctly implement the initial requirements set by the client, or it can be used to determine if the application will suit another user's requirements.

This thesis is organized as follows. Chapter II defines V&V, as well as methods of conducting V&V on mobile applications. Chapter III describes states, statecharts, and how they can are extended to construct statechart assertions. Chapter IV provides a case study using statechart assertions on a set of non-trivial software requirements. Chapter V provides a summary, limitations, and areas of future research.

II. VERIFICATION AND VALIDATION OF MOBILE APPLICATIONS

A. BACKGROUND

Reza B'Far defines mobile computing systems as "computing systems that may be easily moved physically and whose computing capabilities may be used while they are being moved." (B'Far, 2004). One of the earliest examples of this is the abacus. Calculators are successors to the abacus and are equally as mobile. B'Far states that since mobile systems can become stationary just by standing still, they will include the same system characteristics as any other stationary system. These characteristics are calculations, storage, and the interchange of information (B'Far, 2004). Since mobile devices include these characteristics as well as others, then the characteristics of mobile devices are a superset of stationary devices. These additional characteristics are: location awareness, network connectivity quality of service, limited device capabilities, limited power supply, support for a wide variety of user interfaces, platform proliferation, and active transactions (B'Far, 2004).

Early examples of devices that followed these characteristics were the Psion, released in 1984, which was the first Personal Digital Assistant, and the Simon Personal Communicator, first developed in 1992. The Simon was the first cell phone to include Personal Digital Assistant features. It was not a commercial success, but it included an address book, calendar, games, calculator and more (Sagar, 2012).

The Nokia 9000, released in 1996, included the game Snake and was arguably the first implementation of mobile software to become successful. Other phone designers followed suit with games like "Pong, Tetris, and Tic-Tac-Toe" (Darcey & Condor, 2009).

Since then, users of mobile devices have been clamoring for more applications, both productive and time wasting.

B. WHAT IS VERIFICATION AND VALIDATION

Verification and Validation (V&V) is a software evaluation process to ensure proper and expected operation. As stated by Michael et al., "Verification refers to

activities that ensure the product is built correctly by assessing whether it meets its specifications. Validation refers to activities that ensure the right product is built by determining whether it meets customer expectations and fulfills specific user-defined intended purposes." It would be useful in software development, even if only one of these two steps was used, but "you can derive the maximum benefit by using them synergistically and treating 'V&V' as an integrated definition" (Wallace & Fujii, 1989).

"V&V comprehensively analyzes and tests software to determine that it performs its intended functions correctly, to ensure that it performs no unintended functions, and to measure its quality and reliability" (Wallace & Fujii, 1989). Simply stated, the purpose of V&V is to ensure the software does what it is required to do, and nothing more. It is a process that is best done throughout the software development life cycle to prevent or reduce design or development errors from being found in the testing phase, which can greatly increase development time as well as cost.

The area of V&V has been extensively researched to determine the most effective techniques for ensuring that the software performs correctly in regards to the requirements. Techniques fall into two categories: Static and Dynamic.

Static analysis "examines program code and reasons over all possible behaviors that might arise at run time" (Ernst, 2003). Dynamic analysis evaluates "operations by executing a program and observing the executions" (Ernst, 2003). As described by Ernst, Static analysis would, ideally, examine all possible run-time states of the program. Since, for anything other than the simplest program, this would create a state-space that is too large to evaluate in a reasonable amount of time, a model of the system is created and abstracted to try to generalize the system. This causes the model to be less precise than would be preferred. Static analysis can be performed manually or automated. Walkthroughs, code inspections, and code reviews are examples of manual static analysis (Ernst, 2003). These three types of static code analysis require humans to look at code and possibly step through the code as if it was running, causing this type of analysis to be slow and difficult. There are many static analysis tools available for most programming languages, such as Pylint for Python and LDRA Testbed for Java and C/C++. Most of the

tools merely evaluate the source code for adherence to programming standards. Static analysis can also be used to evaluate consistency of the programming such as adherence to the best programming practices.

Dynamic analysis is potentially quicker because the analysis is performed while program is running and testing for correct behavior can be done in real-time (i.e., it either does or does not perform). As stated by Ernst, this makes tests precise because an abstraction is not needed. This characteristic makes dynamic analysis easier, but it is also limited to the inputs used. As with static analysis, testing of all possible inputs is intractable, so a subset of possible inputs must be used (Myers, 2004). Since the input testing is limited, a representative subset of inputs must be chosen, and the results must be generalized. A concern is that this generalization may not hold over future execution of the software (Ernst, 2003). Dynamic testing is also limited to the detection of failures during runtime and will not locate logic errors or other non-runtime errors (Juristo & Vegas, 2003). Categories of dynamic testing falls into two categories: Structural and Functional.

Structural Testing, known as Glass-box or White-box testing, conducts testing to evaluate the correctness of the software implementation with full knowledge of how the code is written. If done by a third party, this requirement can be problematic because the code of many software programs of interest is unavailable, typically because it is proprietary. Even if the code is available and the evaluators completely understand the code, tests are typically written from that understanding and are still subject to mistakes and missed test cases. Structural Testing is not useful when it comes to validation because it does not determine if the software is doing what the requirements state, nor does it check if it does only what the requirements state.

Functional Testing, known as Black-box testing, conducts testing without knowledge of the internal structure. Testing is done by considering what the software product is supposed to do, and then evaluating if the expected output is produced for a given input. This technique is a useful way to conduct the validation part of V&V by checking if the software is correctly implementing the requirements for the software. The problems that can arise from using this method solely are that trust that the software is

doing only what is supposed to do is low. This is the verification part of V&V. Other mechanisms would need to be in place to ensure that the software is doing only what the requirements state. Also, developing metrics to quantify the software can allow for gauging the quality of the software. An example could be error rate.

An additional method for verification is the Formal Methods (FM) technique. FM consists of "mathematically based languages, techniques, and tools for specifying and verifying systems" (Clarke & Wing, 1996). The verification role that FM performs is enabled by the use of model checking and theorem proving (Clarke & Wing, 1996). These two techniques are not code inspection, but they do determine if the model on which the system is being built, or any theorems used are correctly developed and implemented. While there is never a guarantee that every flaw will be found, if the proven models are correctly mapped to the code, the assurance of proper operation of the software is much higher (Clarke & Wing, 1996). The negative side of using FM is that it takes extra time and money to do such a thorough code evaluation in order to construct the correct models or theorems for any non-trivial software. This technique also does not sufficiently cover other aspects of code correctness. It is not likely that the source code for the applications the DoD uses will be available, and is not likely that the DoD will want to pay and wait for such an analysis except when security is a concern. Also, if a different set of requirements were applied to the same piece of software, the previously conducted formal models would not be sufficient.

From the brief overview above, the reader will see that V&V is a complex mechanism for determining the correctness of software and that no single testing technique is suitable for handling both verification and validation. This thesis does not attempt to do so, but does provide a method for performing functional testing

C. DIFFICULTIES IN TESTING MOBILE APPS

New mobile devices, especially phones, have such short development times that the devices have barely been on the market long enough to work out existing bugs before the new device with new software is ready to release. As an example, Apple releases a new iPhone model every year, and has developed six generations of iOS. The Android

operating system had eight versions in three years. This high turnover of phones is created not only by demand and competition, but also capability increases of computing power, battery life, and screen size (The Evolution of Cell Phone Design Between 1983–2009, 2009). As new capabilities are added to the devices and applications in each development cycle, new automated testing techniques are needed to keep up with the fast pace of mobile application development.

Additional difficulties in testing of mobile applications are due to limitations of the hardware. At this time, other than operating system tasks, iPhone can only run a single application at a single point in time. The purpose is to conserve the limited computing power of the device as well as reduce power consumption. The negative aspect is that there is little or no application interaction on a single device. This prevents useful testing applications from running on mobile devices to analyze the real-time behavior of application. Even if such ability were possible, the small screen size would create difficulties in analyzing the data while on the device. Android devices have the ability for third party developers to create multiprocessing applications, which could allow analytics to be conducted directly on the device, but the same screen size limitation would impede analysis of the data.

These limitations make testing done off the device more amenable. There are two possible options: use device-specific emulators, or use specially altered software code to allow offloading of real data from the device onto a computer for analysis. While the emulators will do a good job creating a proper environment to test an application, it has the limitation of being stuck in place, and does not recreate the ever-changing environment where mobile devices exist. The other method could potentially include such a robust environment; the currently existing techniques require a cable connection to a computer tethering the mobile device to an immobile one. The current techniques also require an instrumented version of the original code to provide a mechanism to offload the required information to properly evaluate the operation of the application.

D. CURRENT SOLUTIONS TO V&V OF MOBILE APPS

Bo, Xiang, and Xiaopeng (2007) and Muccini, Francesco, and Esposito (2012) have done extensive research regarding the state-of-the-art of software testing of mobile applications. Muccini et al. did a great job identifying the differences between context-aware mobile applications, other types of mobile applications, and traditional applications. In doing so, they listed several topics that are normally tested for software applications, and provided what they consider as best solution. A brief overview of current tools and techniques for testing mobile applications will clarify what is currently available, as well as their weaknesses.

Monkeyrunner enables the writing of unit tests to test software at a functional level (monkeyrunner, n.d.). Monkeyrunner uses Python to run testing code on one or more devices, or an emulator. It can send commands, keystrokes, and record screenshots. Monkeyrunner allows for repetition of test results, but element location in the recorded screenshots is the basis for comparing two test results. This limits comparisons to a single screen size.

Android Robotium is a Java-based tool for writing unit tests (Robotium, n.d.). Similar to monkeyrunner, it is designed to run as a black-box testing tool and also can run as an emulator as well as run on the actual device, although limited to a single device. Robotium allows for testing of pre-install software as well. The big difference between Robotium and monkeyrunner is that Robotium has a more robust test result comparison. Rather than using a location-based method, Robotium uses identifiers to recognize elements. This allows devices of different types and sizes to be compared to ensure consistency.

Lesspainful.com provides a way for customers to run software and unit tests on physical devices without cost of owning the devices. The customers use the programming language Cucumber to write an English description of the test they would like to run on their software. Once the devices to be tested on are chosen, the tests are automated in a cloud-like system with results from each mobile device presented to the customer to allow for easy comparison.

Testquest 10 is a software suite, created by BSquare, which enables unit tests in a device emulator and enables collaboration of geographically dispersed teams (BSQUARE, 2013). It utilizes extensive use of image recognition to determine device state as well as the location of applications and features on the screen. An interesting feature is that if the GUI design is changed and an application or feature is moved from one location to another, this suite is able to locate and use the feature.

Bo, Xiang, and Xiaopeng (2007) introduce an approach for testing a device and software by using what they call sensitive-events. Their approach reduces the need for screenshot comparisons by capturing these events, such as inbox full, to determine state change. The software will then evaluate these state changes and if the events indicate desired conditions, the tests will continue.

All of the aforementioned software tools are for testing an application to ensure proper functionality and operations. What they are missing is the ability to map the operation of the phone directly to a set of requirements. The above tools all require some form of script writing, although Lesspainful.com makes it simpler, which can introduce missing software test cases. When writing scripts to cover unit tests, the programmer must understand the requirements and determine boundary (edge) cases and properly test for them. The tools are also limited in their ability to handle context-aware features. Most software tools do not handle the context that the phone is in and are generally limited to the location of the text environment. Another limitation is that due to the limitation of the hardware and the software testing suites, only one application at a time can be tested.

Delamaro, Vincenzi, and Maldonado (2006) use an extension to the JaBUTi, called JaBUTi/ME. The extension takes JaBUTi, which is a Java byte code analysis tool, and adds the ability to run instrumented-code on a mobile device that creates trace data, and then pass the trace data to a desktop computer for analysis. By using a method of creating trace data, this solution is conceptually similar to the idea presented in this thesis. This method is still limited by requiring test cases to be written to evaluate the resulting trace file. Additionally, as stated by the authors, the code instrumentation would vary based on the hardware device the code is being tested on. This is due to the potential differences in network connectivity needed to transmit the trace data back.

E. NEW METHOD FOR MOBILE APPLICATION V&V

The current methods, as discussed in the previous section, all require a thorough knowledge of the operation of the application, both proper and improper. That way, the evaluation tests can be written to test that the device acts correctly when given both good and bad input. This thesis puts forth a method that will not only allow testing of the application in the expected environment of operation; it also makes the creation of evaluation tests to be much simpler.

Our technique provides a mechanism to conduct V&V based on log files generated, during or after development, by an application undergoing analysis. Using log files produced by an application brings a couple of benefits: 1) it captures the behavior of the application on an actual, physical device and 2) the data contained in the file will represent the behavior of the application as it executes. The latter benefit is especially important. Whereas other methods of evaluating software mentioned previously involve running software on a device connected to a computer in a static or limited mobility environment, log files collected by the application in execution on a device that is fully mobile mean the log file could hold data that is representative of expected normal operation of the application. Therefore, the device with the application could be used in its normal operational environment and then the log files could be analyzed to determine if the behavior was correct based on the requirements.

Using a method for representing software requirements as statechart assertions, we are able to utilize the log files to verify the correct behavior of the application that generated the log files.

F. STATECHART-BASED V&V

Statecharts are “a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e., concurrency) and refinement, and encouraging ‘zoom’ capabilities for moving easily back and forth between levels of abstraction” (Harel, 1987). As Harel explains, statecharts were developed because of “the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and rigorous…” (Harel, 1987). Harel describes telephones and

communication networks as reactive systems, which are event-driven systems that react to external and internal stimuli (Harel, 1987). This fits well for our use with mobile devices. The limitation of Harel statecharts is that they model the operation of a system from the system's perspective.

Because of that limitation, Dr. Doron Drusinsky extended the capabilities of statecharts, called statechart assertions, to allow for the creation of formal logical statements of requirements from the customer's perspective. Statechart assertions also allow for the creation of simpler evaluation test cases. The current methods, as described in the previous section, all require development of tests to cover all possible situations to determine if the software operations correctly, or a subset of these situations to gain some assurance of correctness. Statechart assertions allow for test cases to be developed to represent the expected operations as dictated by the software requirements. A single statechart assertion can represent one requirement; thus complexity of a single statechart assertion is only as complex as a single requirement. Any operations that violate the behavior specified by the statechart assertion are incorrect. Any operations that are deemed incorrect by this method, but should not be, indicate errors in the requirements.

The use of statechart assertions allow for a reversal of the typical method of testing. The testing software mentioned in the previous section all create tests and then run the tests on the emulator or device to determine if the device is working properly. The technique presented in this thesis uses mobile device application behavior (output) to be passed into one or more statechart assertion to ensure they meet the requirements. This allows for a single set of captured behaviors to be used multiple times during statechart assertion refinement, or if the software is being considered for different sets of requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

III. USE OF STATES, STATECHARTS, AND STATECHART ASSERTIONS

This thesis describes a technique to conduct mobile application V&V that provides both a mechanism to test the application in its intended usage environment, as well as enable straightforward verification of the software's behavior with its defined requirements.

To allow for testing in the intended environment, we shall use log files that store the information about the software while it is operating. Since the thesis is based around software development for the DoD, a requirement for the software can be that a log file must be kept to store the necessary information. This log file will then be used to generate an executable description of the software's operation that can be compared with the stated requirements to ensure that it is operating as defined.

Statechart assertions will be used to describe the software requirements and will enable automated verification of the software behavior. David Harel originally developed statecharts for use with complex reactive systems. A reactive system is event-based and constantly must change based on both internal and external triggers (Harel, 1987). Statechart assertions extend statecharts to enable one-to-one mapping of a statechart to a software requirement (Drusinsky, Michael, Otani, & Shing, 2008).

A. STATES AND EVENTS

The use of states and events has been one of the primary methods for describing reactive behavior of systems, both simple and complex. Creating a finite state machine (FSM) and its corresponding state-transition diagram provides a change-by-change flow showing the operation of a system. Each state represents the condition of the system at that level. If any condition of the system is different, such as a mouse cursor move in a computer system, then the system transitions to a different state. States and events are effective in describing a system, but because a state in the FSM must exist for every possible state of the system, a complex system is subject to state-space explosion. Even

though the number of states is finite, a moderately sized system is still complex enough to make it extremely difficult to analyze for correctness.

B. STATECHART FORMALISM

David Harel extended the concept of states and events to compensate for tendency of state-space explosion in reactive systems by changing the requirements that each state much be represented explicitly (Harel, 1987). The results are statecharts, which provided this ability as well as others. Harel (1987) described statecharts as "a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality and refinement, and encouraging 'zoom' capabilities …"

Statecharts uses an enhanced state-transition format as the state/event method. See Figure 1. The general form is: Begin in State 1. If event 'a' occurs and if condition 'b' is true, then transition to State 2.

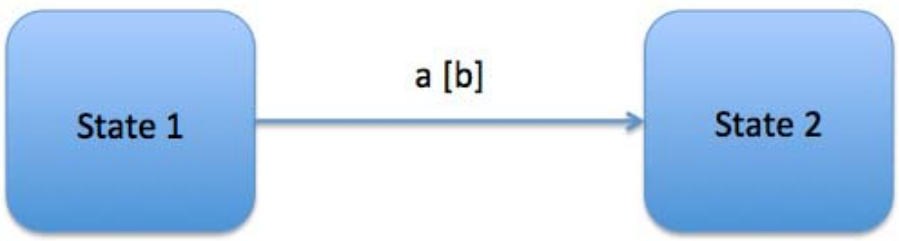

Figure 1. State/event transition

This captures all the needed formality of a state/event chart. If we assume that each state has a possible transition to every other state in the system, we can begin to see how the state-space explosion can occur. With two states, we have a maximum of four transitions (2^2), if we include transitions back to the same state (i.e., State 1 -> State 1 transition). A system with three states will have nine transitions (3^2). A system with four states will have sixteen transitions (4^2). Although systems will not always have transitions to every other state, it is evident how difficult managing transitions could become in a complex system.

As seen in Figure 2, the format for the basic state-transition format is very similar to the State/Event format as seen in Figure 1. The A following the forward slash above the arrow indicates an action that will take place upon a valid transition. Actions can also be utilized in states as a feature upon entry, exit, or both. Actions have an associated activity that is a physical expression of an action. As stated by Harel, activities take a nonzero amount of time. As an example, the action may be <start> and the activity may be <powerOnComputer>. The end result is a computer that gets turned on. The action A in Figure 2 indicates such an expression. A timer feature was also added to the statechart implementation to allow for state transitions or action to occur after a set interval.

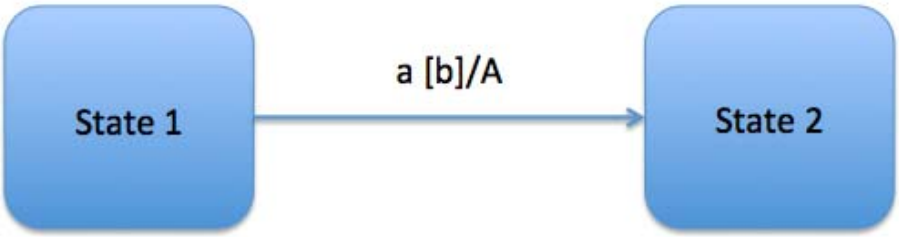

Figure 2. Statechart transition

There are many other features that statecharts include, such as superstates, history capability, and conditions and selection entrances, these are not applicable to this thesis and shall not be discussed. For further information on the history feature and other details on superstate features, please see (Harel, 1987). Orthogonality, the ability for two subsystems to run simultaneously and independently, as described by Harel is also not used in this thesis, but the concept will be used in a different form.

It is interesting to note that when Harel was analyzing the Timex watch as an example, he was able to find flaws in either the original requirements, or the software design phase. This is exactly what the technique this thesis proposes is attempting to do.

C. STATECHART ASSERTIONS

The statechart assertion extends Harel statecharts, or modeling statecharts, by adding a 'bSuccess' Boolean flag and by enabling non-determinism (Drusinsky, Michael, Otani, & Shing, 2008). Statechart assertions are formulated from an external observer's perspective. Though the bSuccess Boolean is a simple mechanism, it is instrumental in determining if an assertion ever fails. The Boolean indicates whether the assertion was violated by the system being analyzed. A statechart assertion assumes the requirement it is based on is met (bSuccess = true) and it will retain that assumption unless a sequence of events leading to the violation of the requirement specified by the statechart assertion is observed. Once an assertion fails (i.e., reaches an error state), bSuccess becomes false and will stay false for the remainder of the execution. Since the statecharts are simple, it is easy to identify the assertion that failed and the cause.

Using requirement 1, we can generate the statechart in Figure 3. Starting out in State 1, if event a occurs with the condition b being true, then the Error state will be reached. The entry action for the Error state sets bSuccess to false, meaning that the requirement Figure 3: Simple statechart assertion is based on is not met.

Requirement 1: Event a must never occur.

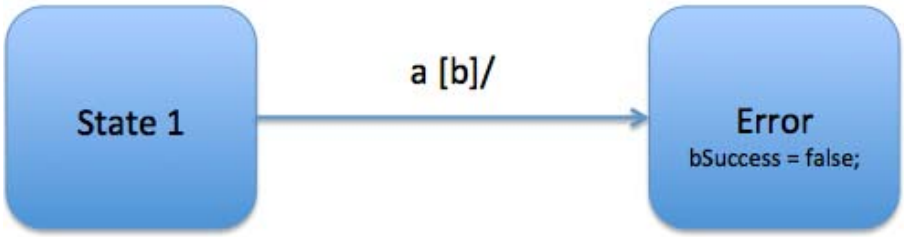

Figure 3. Simple statechart assertion

As described in the report "The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors," a model-based specification uses a single, intertwined representation of the software requirements (e.g., as a single statechart), and as a result can become complex to understand the interaction of each requirement with

others. An assertion-based specification allows the requirements to be decomposed into their simplest forms, and then create a formal representation (e.g., a statechart assertion) for each requirement. This decomposition allows a one-to-one connection between a statechart assertion and a customer requirement. A significant benefit of this connection is that it simplifies the development, analysis, and testing of the statechart assertions. Ideally, the set of statechart assertions is the same as the complete set of customer requirements. In addition to these benefits, Drusinsky, Michael, and Shing (2007) state:

- Since the complexity of the statechart assertions is minimized, any changes to the requirements will simplify the changes that need to be made to the assertions.
- Statechart assertions can be made to represent a test for negative behaviors where other assertions can stay as tests for positive behaviors.
- Tracing unexpected behaviors to the one or more requirements it is violating is simpler since there is a one-to-one mapping.

Drusinsky also extended Harel statecharts to include non-determinism. Since the statechart assertions are non-deterministic, for a given input and state of the system, there can be zero or more transitions from a state. This is important because more than one instance of a requirement may be evaluated concurrently during the execution of the software.

Multiple instances of a given requirement provide the real reason non-determinism was included into statechart assertions. It allows the assertions regarding timers in systems to be more robust. A timer can function in a system in two ways, either in repetition, or in overlapping instances. Non-determinism is necessary to enable overlapping instances. To provide a more illustrative description, let us define a new requirement.

Requirement 2: No more than 2 Invalid Password entries within 15 seconds.

Without non-determinism, a statechart would be limited in its ability to handle timers. Figure 4 is an easy initial attempt to capture Requirement 2. When an Invalid Password occurs, a timer starts and a transition to State 2 occurs. If another invalid entry occurs, the machine transitions to State 3. If a third Invalid Password occurs, then the system has violated the requirement. Otherwise, if the timeout occurs, then we reset and start again. This seems to satisfy the requirement, but if we examine the timing diagram in Figure 5, we see the problem is that the statechart is only able to handle a single fifteen-second window beginning at the first Invalid Password entry. The second Invalid Password should begin a separate fifteen-second window while the first one is running. As we see in the Figure 5, a violation of Requirement 2 occurs, but is undetected.

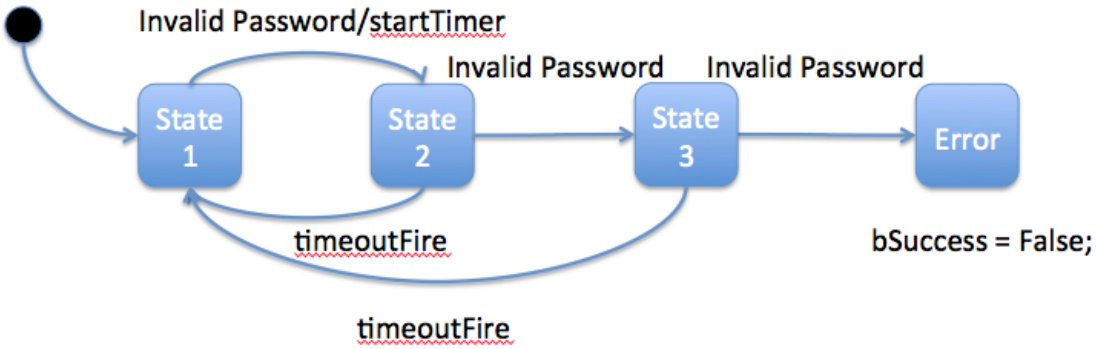

Figure 4. Deterministic example

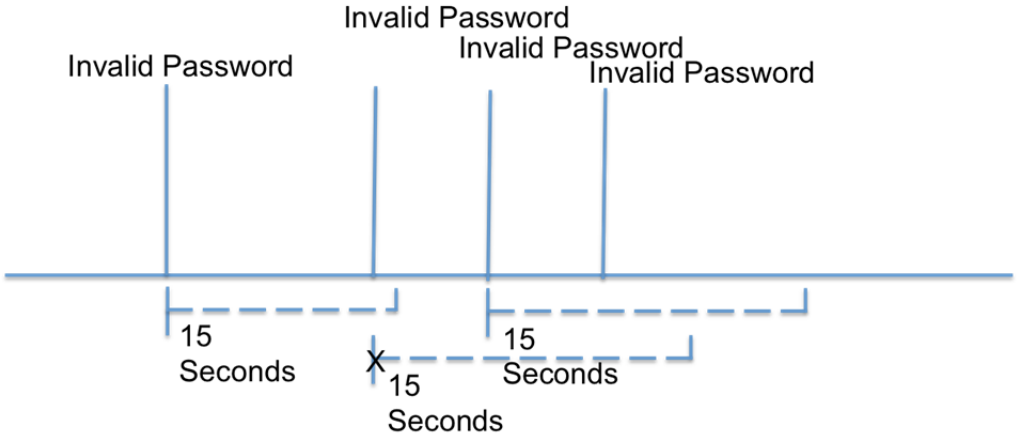

Figure 5. Timing diagram for Figure 3.

Figure 6 shows a better statechart assertion implementation of Requirement 2. If an Invalid Password is received, timer 1 starts and transition to State 2. If a second invalid entry is received, the system starts a second timer and transitions to State 2a. If a third is received, then the system transitions to the Error state, in which bSuccess is set to false meaning the requirement failed. While this is straightforward, the handling of the timers complicates the diagram. If, in State 2a, the timer ends for the first invalid entry before a third could be received, the system transitions to State 2b. If a new Invalid Password is received, the system starts timer 1 again and transitions to State 2c. If the second timer ends, then transition back to State 2. This diagram seems to work correctly according to the timing chart, but the problem is complexity. For this example, the statechart assertions require a new state for every combination of timer conditions. Each combination of running/not running must be accounted for as well as which one has less time left. State 1 has neither timer running. State 2 has timer 1 running. State 2a has timer 1 and 2 started with timer 1 with less time. State 2b has timer 2 running only, and State 2c has timer 1 and 2 started with timer 2 having less time. If the requirement was expanded to allow 3 invalid entries within 15 seconds, we would require an extra timer meaning we would have several more states as well as a much more complex interaction of states.

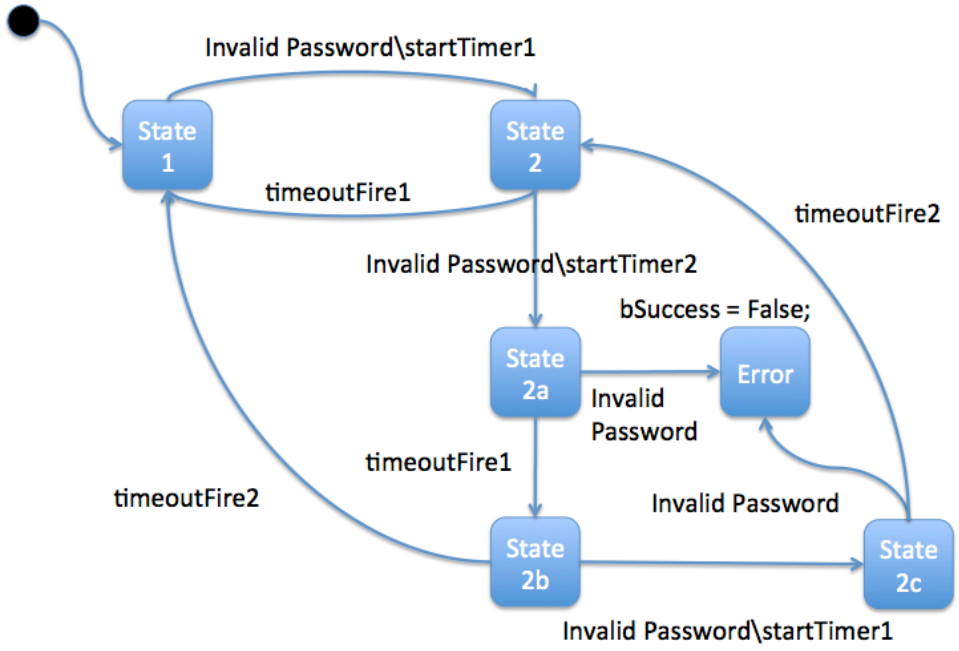

Figure 6. Complete deterministic example

Figure 7 is a non-deterministic statechart assertion based on Requirement 2. As can be seen, the diagram reduces the complexity and can offer as many timers running at the same time as needed. Notice that there are two transitions from State 1 on an Invalid Password. For each Invalid Password event received, an instance of a timer is started that is orthogonal from any others that may be started and transitions to State 2. For each subsequent Invalid Password event received, a new instance is created and transitions occur in any instances that have been previously started. If two more Invalid Password events for a given instance are received, the system will transition to Error state, ending the instance in error. If a timer expires before Error state is reached, then that instance will end in the Success state. If any timer instance ends in the Error state, the assertion will have failed. This characteristic makes the non-deterministic timers operate as an AND operation where every instances must be true. As mentioned before, non-determinism simplifies changes that are made. For example, to expand the system to allow up to three invalid entries within fifteen seconds all that is necessary is to add one state between States 3 and 4, and an arrow flowing from the new state to the Success state. This is much simpler than making changes to Figure 6.

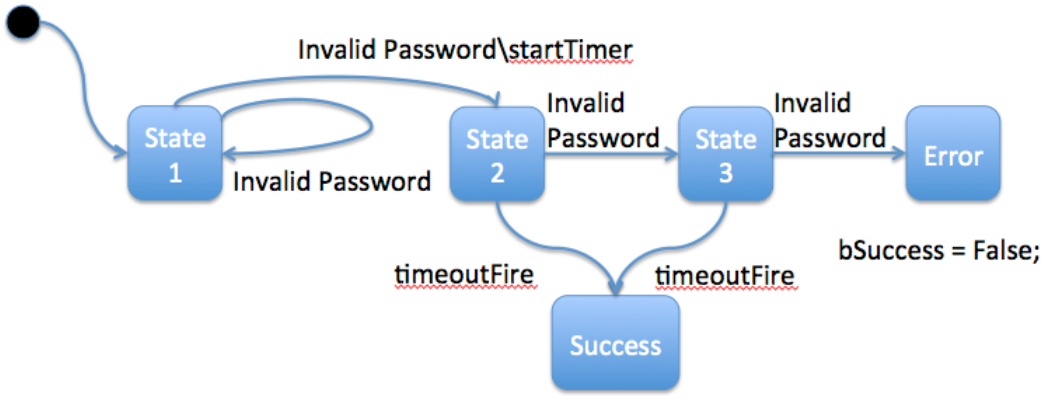

Figure 7. Non-deterministic example

Figure 8 shows the timing diagram for both figures 6 and 7. This demonstrates both the deterministic and non-deterministic statecharts function identically.

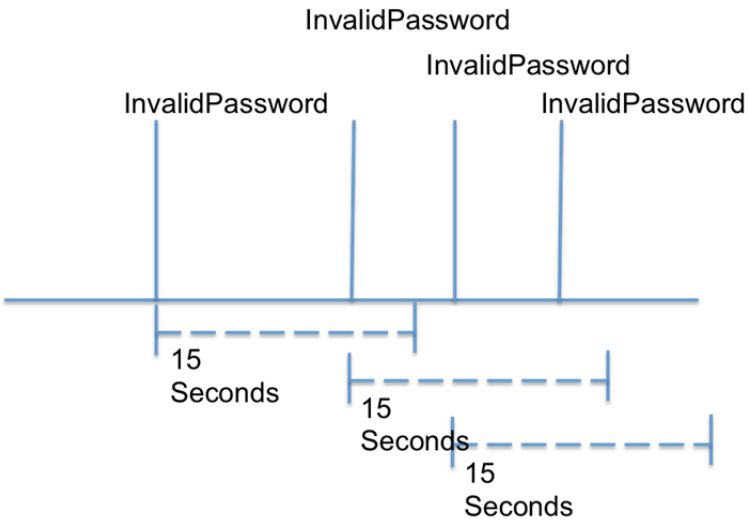

Figure 8. Timing chart for Figures 6 and 7

Statechart assertions allow for a reversal of the typical method of testing. The testing software mentioned in the previous chapter creates tests and then runs the tests on the emulator or device to determine if the device is working properly. This requires development of tests to cover all possible situations to determine whether the software operates correctly, or a subset of these situations to gain some assurance of correctness. Statechart assertions enable a mobile device application behavior (output) to be passed into one or more statechart assertions to ensure they meet the requirements. This allows for a single instance of mobile device behaviors to be used multiple times during statechart assertions refinement, evaluation for correctness, or if the software is being considered for different sets of requirements.

D. LOG FILE-REPRESENTED BEHAVIOR

Log file are useful as they show behavioral details about an entity during operation. This could refer to the log of a guard post, or an automated door, or software. Log files created by a software application during operation in its normal environment presents the most useful data because it shows how the application operates within, or

interacts with its environment. This enables testing of the application to represent real-world characteristics, not just what can be produced in a lab environment. Using log files that represent the behavior of an application and statechart assertions that can be used to verify behavior against requirements, an application shown to meet the requirements set forth by the customer.

The log file will contain all the pertinent information regarding the behavior of the software, (i.e., events that occur in the application). The logging functionality must be coded into the software during development and should be a requirement for DoD software. Since it is the responsibility of the software developer to produce the log file, the software tester only needs to collect the log file.

To facilitate the evaluation of an application using a log file, we use the StateRover tool. StateRover enables the development of statechart assertion by extending the TimeRover module that is available with the Eclipse Integrated Development Environment. StateRover uses Java Unit (JUnit) tests to exercise the statechart assertion. TimeRover based on a log file automatically generates the JUnit tests. For this automatic generation to be possible, the log file must follow a specific format.

1. Log File Format

An example of a formatted log file is shown in Figure 9. All spaces are for readability only.

```
<newtest>
<event>
<sig><![CDATA[main()]]></sig><time lang=“c” unit=“sec” val=“time1” /></event>

<event>
<sig><![CDATA[function1(double arg)]]></sig><args><arg type=“double” name=“arg”
/></args><arg0 val=“value1” />
<time lang=“c” unit=“sec” val=“time2” /></event>

<event>
<sig><![CDATA[function2()]]></sig><time lang=“c” unit=“sec” val=“time3” /></event>

<event>
<sig><![CDATA[function2()]]></sig><time lang=“c” unit=“sec” val=“time4” /></event>
```

Figure 9. Log format

The file starts with a <newtest> marker to indicate the beginning. A new event is indicated with an <event> tag begins a new event. A </event> indicates the end of the event. A <sig> tag marks the beginning of the event that is occurring. The information in the tag provides information on the name of the event. In the preceding example, the functions are main(), function1(double arg), and function2() in that order. Depending on the type of event that is occurring, the next tag could be the <args> tag, which provides arguments to the event. Following the <args> tag, each argument that is provided requires the value of the argument. Following the argument entry, information about the timing of the event is included. Line two has a single argument of type double and named 'arg'. The value is 'value1.' If there is not an argument field, then the timing information immediately follows the <sig> tag. In the example, time is represented by time1 through time4.

2. JUnit Tests

JUnit tests are the Java variant of unit tests used to determine if code works as expected by exercising the functions using manually created code. The typical method is to assert that some activity occurred causing a function or method to be executed, and then assert that some condition is true or false. If the unit test succeeds, then the tested code operates as expected. If the unit test fails, then the code is not correct. In the case of statechart assertions, the events captured in a log file could be a call to a method or function, such as an 'Invalid Password' event in Figure 8. An example condition to test if true or false would be to assert that the timer did start as expected. If it did, then the software is successful. If not, an error occurred. Another condition check is to determine whether the bSuccess Boolean is true or false. In fact, this is how we determine success or failure of the log file in our case study.

The case study in Chapter IV will use a log file collected from the application that contains data not presented in the format required. We will use a program that converts an application log file into a format shown in Figure 9. Needing to convert an application log file may not be necessary if the required format is provided to the app developer. The

developers of the application will ensure the application log file output would follow the provided format. There are reasons why this may not be the best way to go:

- The app log file could provide more information than needed, but the tester only needs to use the information needed. But if the needs change, then the data is already available.
- The format for the test is not provided, thus possibly keeping the intent and desires of the test a secret from the developers of the application.
- The nomenclature of the statechart assertions probably will not match with that of the log files, thus some number of changes will probably be necessary.

E. DEVELOPMENT OF STATECHART ASSERTIONS

As mentioned previously, a one-to-one mapping from statechart assertion to customer requirement is the goal. To properly develop a statechart assertion that accurately represents the natural language requirement provided by the customer is a significant concern. According to Michael, Drusinsky, Otani, and Shing (2011), development of any product requires a consistent working relationship between the customer and the developer. A tightly controlled validation and verification process, shown in Figure 10, can aid in the development of the statechart assertions.

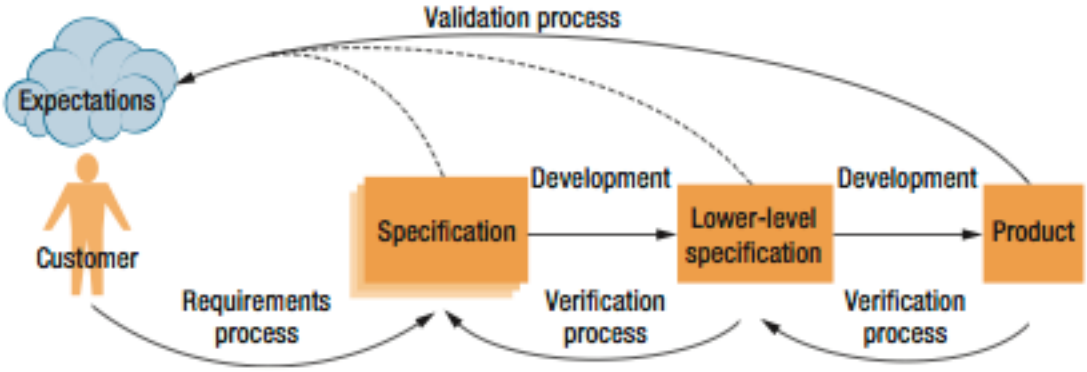

Figure 10. A continuous validation and verification process (From Michael, Drusinsky, Otani, & Shing, 2011)

Even though in our case the product is a correct statechart assertion, adhering to this process will provide faster and more complete development of the assertion.

1. Requirements Gathering and Statechart Assertion Development

Performing V&V using statechart assertions starts out the same as many other methods of verification. The idea is to gather the requirements from the customer and create statechart assertions that correctly implement those requirements. This is the most time-consuming part of the process, but every verification method must go through the same process. It requires careful consideration of what the customer states. It also requires understanding of the need and the customer's current solution to ensure the generated requirements are complete. A subset of these requirements, particularly the ones that are mission-essential and safety-critical, will be formally captured using statechart assertions. Once the assertions are generated, the customer will need to review and understand so that the resulting assertions can be approved. There are books available that provide a better understanding of requirements gathering and verification such as (Wallace & Fujii, 1989). Harel (1987) does an excellent job describing how to develop statecharts from requirements. The desire is to create requirements that are as simple and complete as possible. This will make the generation of a statechart assertion simpler. A good idea is to have only one characteristic of the system tested along with only one mode of failure. This necessitates the development of more statechart assertions, but it will mean that changes made later will be simpler to perform.

2. V&V during Statechart Assertion Development

Figure 10 shows that validation is a customer-centric process whereas verification is a developer-centric process. Figure 11 shows the process from what could be a step following the requirements gathering phase. Using scenarios developed with input from the customer to simulate a possible environment for the application, JUnit test code is developed to exercise the statechart assertions. If they succeed, then the assertions have captured behaviors as expected by the customer. If they fail, an assessment will be done to determine whether the statechart assertion is incorrectly developed, or whether the requirement the assertion is based on is not correct as expected by the customer. This process is designed to ensure what is developed is what the customer wants.

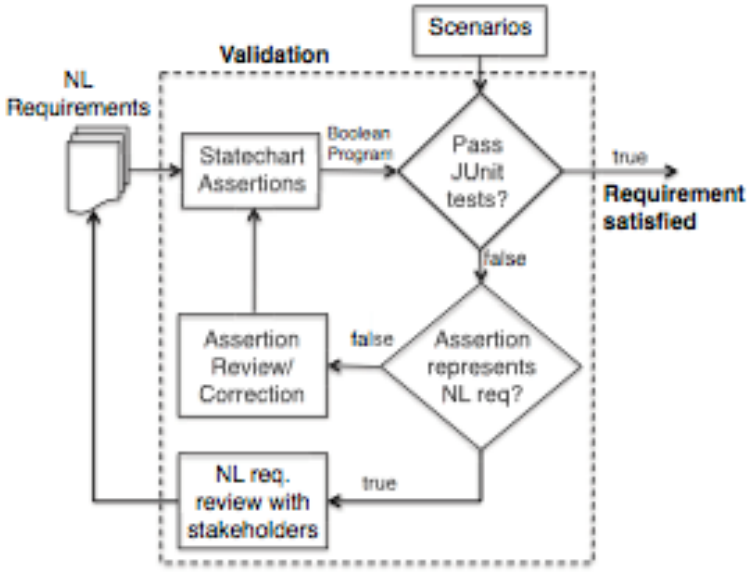

Figure 11. Validation process (From Bergue Alves, Drusinsky, Michael, & Shing, 2011)

This image could also be used during requirements gathering by changing the 'Pass JUnit tests?' decision point to a verbal walkthrough and the statechart assertions block to Natural Language Requirements. By talking through the expected results based on the requirements, the developers can ensure they understand as well as possible what the customer desires.

Figure 12 shows a notional process to verify that what is being developed is what was requested. Using JUnit tests created from log files, either manually created or developed by an executing application, the statechart assertions are checked to see if the expected test result is received. If the results of the JUnit tests are positive, then the software application satisfies the requirements specified by the statechart assertions. If not, the software application has violated one or more requirements specified by the state assertions. There could be one or more possible causes of the problem. A detailed analysis of the execution trace is needed to determine whether the cause of the violation was due to the wrong behavior of the application or an incorrect statechart assertion. Actions will then be taken to correct the problems and the verification cycle is run again.

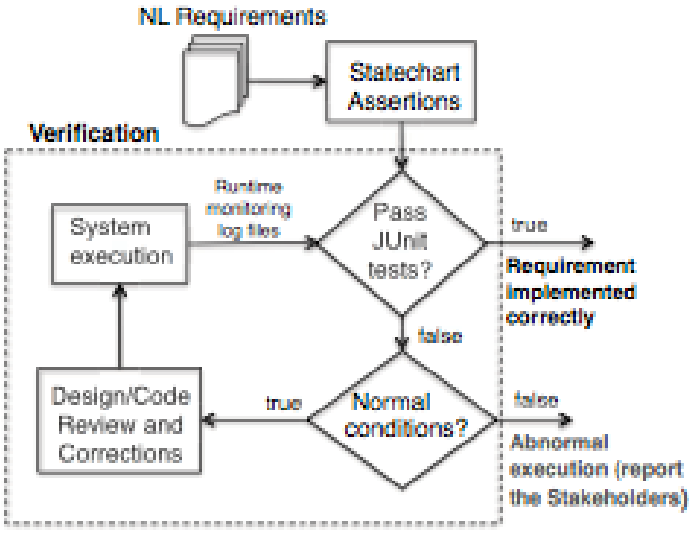

Figure 12. Verification process (From Bergue Alves, Drusinsky, Michael, & Shing, 2011)

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CASE STUDY

The case study is based on a set of requirements for an application that performs tracking of automobile traffic at entrance gates to a naval base and utilizes GPS fencing. Due to the limited capabilities of the application developed so far, we used a subset of the original requirements for this example. In addition to some of the requirements for the application, we will add extra requirements to make the case study more meaningful. Let us consider the fictitious example of a smartphone application that uses a GPS to track the location and speed of a person in motion. A log of the collected GPS data must be kept that will be uploaded to a server. GPS applications can consume a lot of power and storage space and since mobile devices have limited amounts of both, minimizing the consumption of both is important.

Although these requirements are simple and seem to be straightforward, as with any requirement definition, refinement of the customer requirements is necessary. After reviewing the original requirements and requesting clarification, we need to include the following additions. Due to the limited available storage space on the mobile device we must minimize the amount of GPS data stored. The method chosen to accomplish this is to adjust the rate at which the GPS updates occur to be based on the speed at which the user is traveling. An additional requirement is that the log file must be able to be transmitted from the device to a server by a Wi-Fi connection only. This is because many of the users will not have wired connectors for the devices. If at any point Wi-Fi connectivity is lost, and there is an active transmission, it must be terminated. The application has a limit of thirty seconds to transmit the log file; after which, if not successful, the user must be notified. The user must be notified of the failed transmission within five second. Additionally, a log file must not be transmitted within one hour of a previous log transmission. Both the use of a time-limited transmission window for the log file as well as an infrequent upload of the log file will aid in reducing the amount of power and bandwidth the application consumes.

These additional requirements provide better detail about the customer's expectations and will allow for quicker development of the statechart assertions. We will break the requirements down into groups based on speed-based GPS updates, log transmission, and Wi-Fi-connectivity.

In our example, we use a log file produced by the application and evaluate it against several statechart assertions, but in order to do so, we need to convert the original log into a log that can be read by the StateRover tool. We use a Python script to convert the application log file into what we shall call a StateRover log file. This is a specially formatted log file that will be described later in the chapter. The StateRover log file is imported into Eclipse and by using a namespace map, a connection between our event listed in the StateRover log file to one or more events in the statechart assertions is established.

A. SETTING UP THE ENVIRONMENT

We are using Microsoft Windows 7 Professional and Eclipse version Indigo to create and evaluate the statechart assertions for this case study.

After downloading and starting Eclipse, the standard setup for Eclipse will be shown. Before building statecharts, a small amount of configuration needs to be performed and an Assertion Repository must be created. The Assertion Repository is one of the custom features created by Drusinsky. It allows several statechart assertions to be evaluated concurrently for each log entry.

First, the TimeRover product must be installed. Under the Help menu in Eclipse, select Install New Software. The location of the menu is shown in Figure 13.

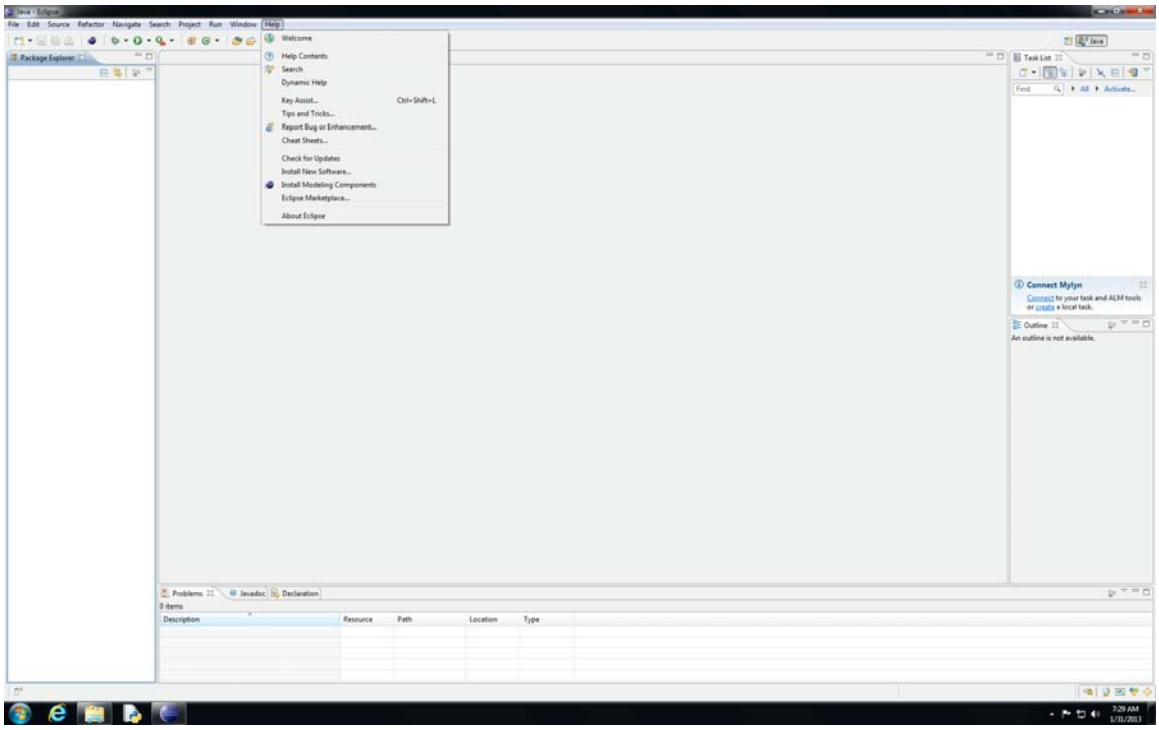

Figure 13. Eclipse version Indigo on Windows 7

Select the Add button, and enter the address http://www.time-rover.com/updates/staterover_Team_3_6 along with a name for the site. A user name and log in will be necessary to download the files. Make sure to deselect the Group Items by Category check box. Select all six files that appear and select Next. Figure 14 shows the files that need to be downloaded. Ensure there are no errors on the next screen, and select Finish.

Once installed, select New, and then select Project. Select the folder Java and then Java Project. Eclipse will ask for a project name. Click the next button, and a screen with four tabs will be shown. These tabs can be used to customize the project as necessary. We need to add the two libraries discussed in Chapter III. Select the Libraries tab, then select the External Jars button. For our setup, the files are located under plugins/ com.timerover.assertionrepositoryjars_1.0.2.201205162118/. Add both the TReclipseAnimation.jar and stateroverifacesrc.jar. After adding them, select the Add Libraries button. This will allow the addition of JUNIT4 test suite. Select JUNIT and then the Next button. In the drop down menu, select JUNIT 4 and click Finish.

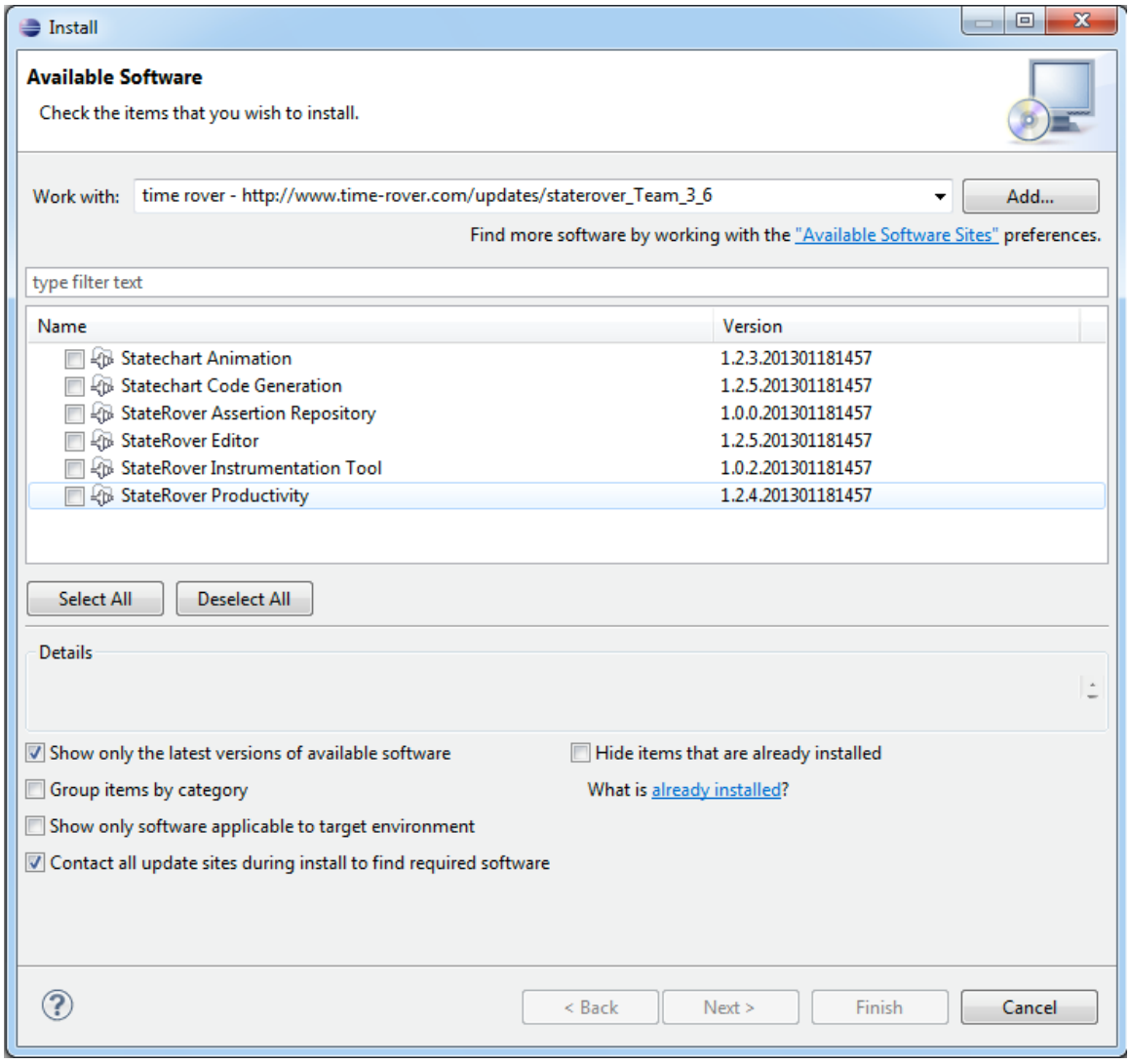

Figure 14. StateRover installation into Eclipse

Once a project has been created, right click on the new project name and select Toggle Assertion Repository. To make the Java code create automatically for your project, right click on the project name again and select Enable Statechart Code Generation. A quick tip is that when creating a new project, the Enable Statechart Code Generation may appear to be already checked, but is not. We suggest you select, and then perform again to ensure that it is actually checked.

Generally, Eclipse will ask if the user would like to view the Statechart Diagram "perspective." A perspective configures your view for the particular type of coding you are doing. This will allow the user to see the properties of statechart elements. Select Yes.

If user was not asked, the perspective can be selected manually. In the upper right, you can see the perspective selector button. Statechart Perspective should be available and should be selected.

Next we need to add a package to the project folder. A package is Eclipse's way of storing all the files needed to create a software program under development. We need to create a special package for each assertion diagram. Right click on the project, and select other. Select the TimeRover folder, and then select Statechart Assertion Diagram. A window will open to ask for a name for the package. Enter your new project name. Ensure the file name ends with ."statechart_diagram." Press next and then Finish.

At this point, the software is ready to develop a statechart assertion. The user may add as many packages in the same manner as needed.

B. SPEED-BASED GPS UPDATES

As stated earlier, the customer would like to ensure the application varies the frequency of GPS updates to conserve storage space. The requirement is as follows.

When a user is traveling at a slow speed like walking, frequent updates are unnecessary since significant distance changes do not happen quickly. If the user is traveling at a faster pace, then more updates allow for more consistent tracking. When the user is traveling at less than or equal to two meters per second, the application should average 5 seconds or more per update. This is approximately the walking speed of a human (Carey, 2005). If the user is traveling at greater than two meters per second but less than or equal to five meters per second, then there must be an average of between 2 and 5 seconds between updates. This will be considered running speed. If traveling greater than five meters per second, then there must be an average of less than 2 seconds between update. This will be driving speed. We decided to use an average of seconds between update due to the typically less than accurate GPS data provided by mobile devices. A requirement for an average over a minimum of five GPS update events will be included to reduce the effects of any lack of precision in the GPS data from the mobile device. Table 1 lists the requirements.

Table 1. Speed-based requirements

Speed Category	Average Speed (x) in meters per second	Expected Time between Updates (y) in Seconds per Update
Walking	x<=2	5<=y
Running	2<x<=5	2<=y<5
Driving	5<x	y<2

Since this seems like all one requirement, the first instinct is to generate a statechart assertion that handles all the requirements. Using StateRover, we develop Figure 15, which successfully define all three requirements for speed-base GPS updates.

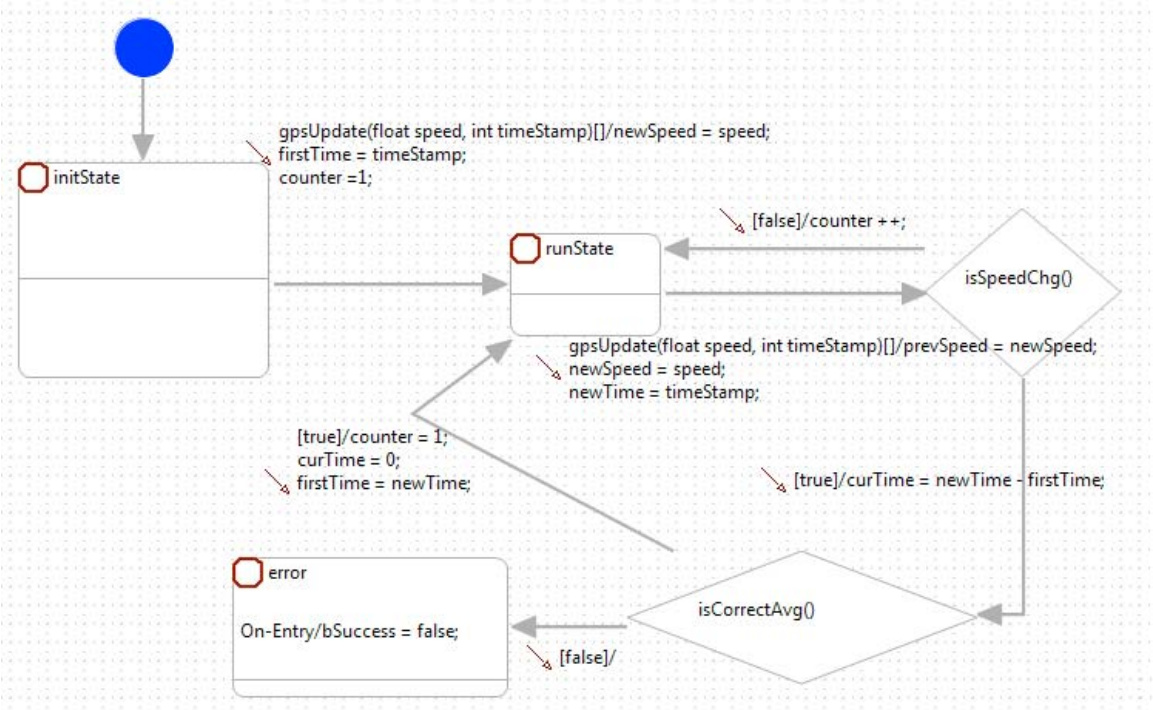

Figure 15. Statechart implementing the speed-based GPS update requirement

At a high level, this statechart assertion will determine whether the speed of the user is in the same speed category as the previous GPS update. If yes, evaluate the next GPS update. If no, then evaluate the last set of consecutive GPS updates in the same speed category. If the average seconds between update meet the requirement, then the

requirement is met. The software will continue to evaluate the most recent GPS update that fell in a different speed category. If the requirement was not met, the application failed the requirement.

Depending on the reader's familiarity of requirement representation with statecharts, Figure 15 could seem complex or simple. We shall refine this statechart into several statecharts later, but first we will discuss this one, as it is completely valid and understanding this will make later statechart assertions more understandable.

As stated in the previous chapter, the symbols used in statechart assertions are the same as those of Harel statecharts. The rounded, white boxes are states, of which we have three, initState, runState, and error. The initState is the starting point for the assertion where no gpsUpdates have been received. The runState is the normal state where the application is "running" other than to check whether the requirement fails or not. The third state is error; if reached, indicates that the application violated the requirement. The blue circle indicates the start point in a statechart assertion. Each diamond is a decision point where a Java statement is evaluated to be either true or false. The result will cause a transition to the appropriate next state.

These functions are used in this statechart assertion:

- gpsUpdate(float speed, int timestamp): An event in the log file that will cause a transition between states.
- isSpeedChg(): Determines whether the most recent gpsUpdate received is within the same category as the previous gpsUpdate. The categories are x <= 2, 2 < x <=5, and x > 5.
- isCorrectAvg(): Determines whether the average number of seconds between gpsUpdates during the previous group of gpsUpdates is correct based on the requirements for that speed category.

The code for these functions is listed in the Appendix B.

The variables used in the statechart assertion are:

- counter: This is used to count the number of GPS updates for a given speed category in order to calculate the average of seconds per GPS update.
- firstTime: Stores the time of occurrence for the first gpsUpdate received for a given set of gpsUpdates in the same speed category.

- newTime: Stores the time of the most recent gpsUpdate.
- curTime: Stores the difference in time between the current gpsUpdate and the first in the series of gpsUpdates for the current speed category.
- prevSpeed: Stores the speed of the previous gpsUpdate to determine if it is the same as the current speed.
- newSpeed: Used to store the current speed to compare to prevSpeed.

We shall describe the statechart assertion using an example application log file. As mentioned previously, StateRover requires the log file to be in a special format, but leaving out the complexity will make this example easier to understand. The required format will be discussed later in the chapter.

0 mps @ 11/30/2012 01:07:29 PM
1 mps @ 11/30/2012 01:07:31 PM
3 mps @ 11/30/2012 01:07:34 PM

Starting at the blue circle, the assertion immediately flows to the initState. Any event received will have no effect other than a gpsUpdate event.

There was a gpsUpdate event at 01:07:29 PM, thus a transition to runState. The speed of the new gpsUpdate is stored as the first in its set. Subsequent gpsUpdates will continue to add to the set if they are the same speed category, or restart the set if they are different. Once the next one is received, transition to the decision diamond. If the function isSpeedChg() evaluates to false, it means that we are still in the same speed category as the previous gpsUpdate. We then increment the counter and transition to the runState. If it is true, the assertion transitions directly to the next decision diamond. In the example log file, the initial state of the system is at 0 mps and the first update is 0 mps, therefore isSpeedChg() evaluates to false. The next gpsUpdate is received one second later. A transition occurs and isSpeedChg() evaluates to false again since it is still in the same speed category as the first gpsUpdate. The third gpsUpdate causes a transition to the first decision diamond, but since it is in a different speed category, the result of isSpeedChg() is true. A transition occurs to the next decision diamond, which contains the isCorrectAvg() function. If it evaluates as true, a transition to runState occurs and resets the variables based on the now previous gpsUpdate. If it is false, there is an error, thus the error state is reached. In this case, we have two gpsUpdates that are in the

walking category within two seconds. The result is 2/2 giving an average of 1 second/update. The requirement for speeds of two meters per second or less is 5 seconds/update or more. This does not meet the requirement, thus we reach the error state.

This statechart assertion will correctly evaluate log files based on the requirements, but there are two issues that make this less than ideal. One is the background code, which is shown in Appendix B. Since the assertion has to keep track of the current and the previous speed, it means there is a little more complexity behind each function. The second is that if there is an issue with a log file, StateRover will notify the user that the assertion failed, but it will not state why. Thus if the statechart assertion in Figure 15 fails, the user will have a more difficult time determining where the failure occurred. To help simplify, we can break this statechart assertion into three discrete ones to help.

Figures 16, 18, and 19 show statechart assertions for each of the three speed categories of the of the speed-based GPS Update requirement.

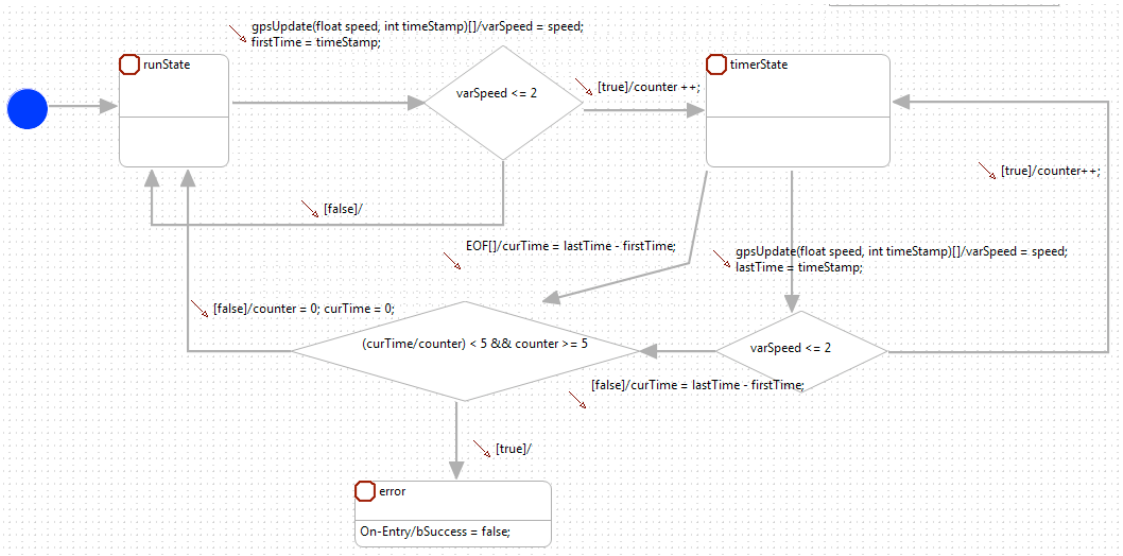

Figure 16. Statechart assertion for speed less than or equal to 2 meters per second

The assertion effectively starts at runState. Once a gpsUpdate is received, flow to a decision diamond. Note that we only store the new speed, and not the previous speed. A

decision is made if the new speed is in the range that this statechart assertion is concerned about. Each of the three statecharts is developed to one of the speed categories. In the case of Figure 16, the walking category is represented. If the decision of on the new speed is no, go back to runState. If yes, increment the counter. When the next gpsUpdate is received, perform the same evaluation of the speed of the gpsUpdate. If it is in the same speed category, increment the counter. If it is not, then determine average seconds per update and ensure there has been the minimum number of updates as required. If true, then return to the initial state. If not, then the application did not meet the requirements. A transition on an EOF event is included to handle the case when the end of the log is reached. Without this, the gpsUpdates at the end of the file not be evaluated. Since evaluation of a set of gpsUpdates for correct seconds per update occurs when a gpsUpdate of a different speed category is received it would not occur at the end of a file without a mechanism to trigger the check.

The only difference between the statechart assertions in Figures 16, 18, and 19 is the code in the decision diamonds. While this refinement of Figure 15 may seem more complex, they do provide some additional benefits. If one fails, it is easier to determine what caused the failure. Also, the background code needed is greatly reduced. Instead of what is used in Appendix B, it is reduced to that shown in Figure 17.

```
float varSpeed = 0;
int counter = 0;
int curTime = 0;
int firstTime = 0;
int lastTime =0;
```

Figure 17. Code for Figure 16

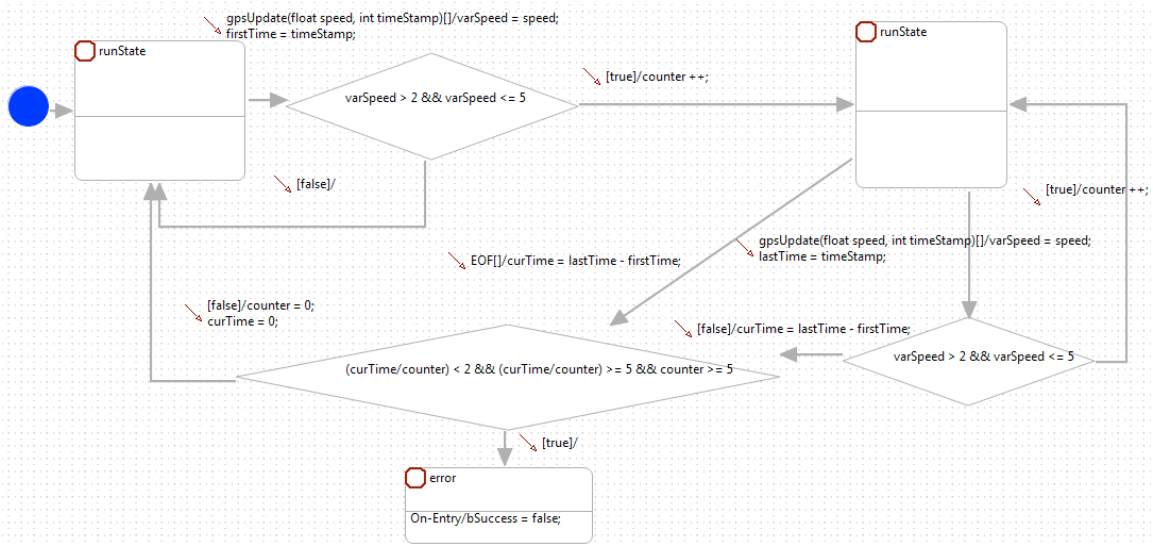

Figure 18. Statechart assertion for speeds more than 2 but less than or equal to 5 meters per second

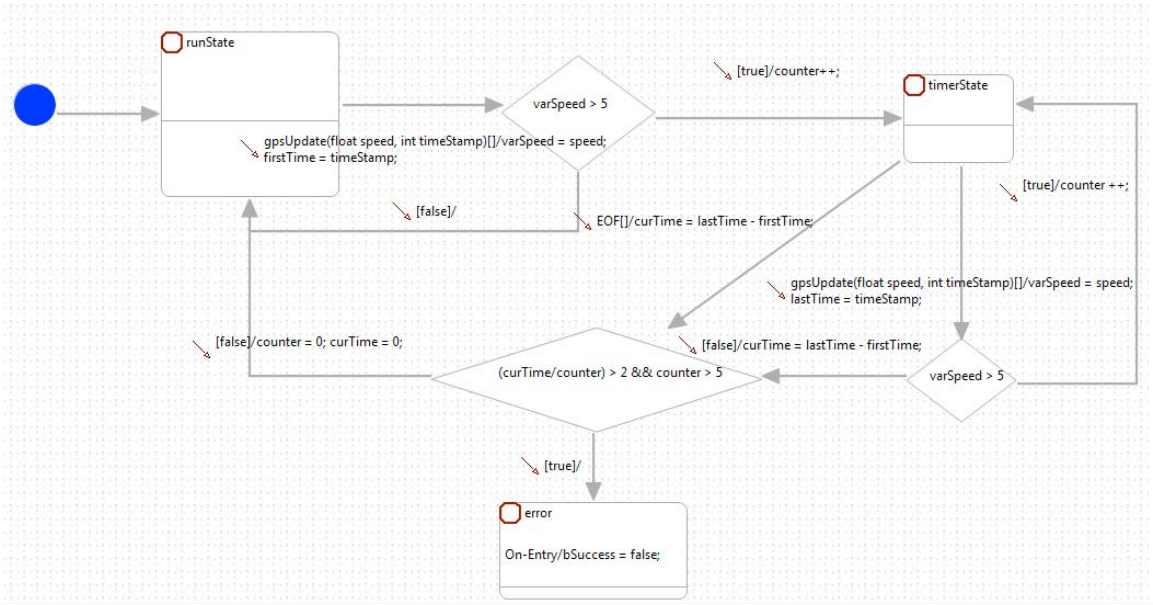

Figure 19. Statechart assertion for speeds greater than 5 meters per second

Even though the complexity is not significantly reduced, an assertion developer only develops one new statechart assertion and the other two are duplicates with minor changes.

C. WI-FI CONNECTIVITY

This requirement states that a log file can only be transmitted when the device is connected to a Wi-Fi access point. Figure 20 shows that when a 'wifiConn' event occurs, then transition to the onWifi state. At this point, an infinite number of transmits will be allowed. If the device is disconnected from a Wi-Fi signal, any subsequent transmits will be evaluated as an error. We know this statechart assertion does not prevent log files from being transmitted within an hour of each other, violating a requirement, but it does not need to. A separate assertion will handle that part, thus allowing a simpler statechart design.

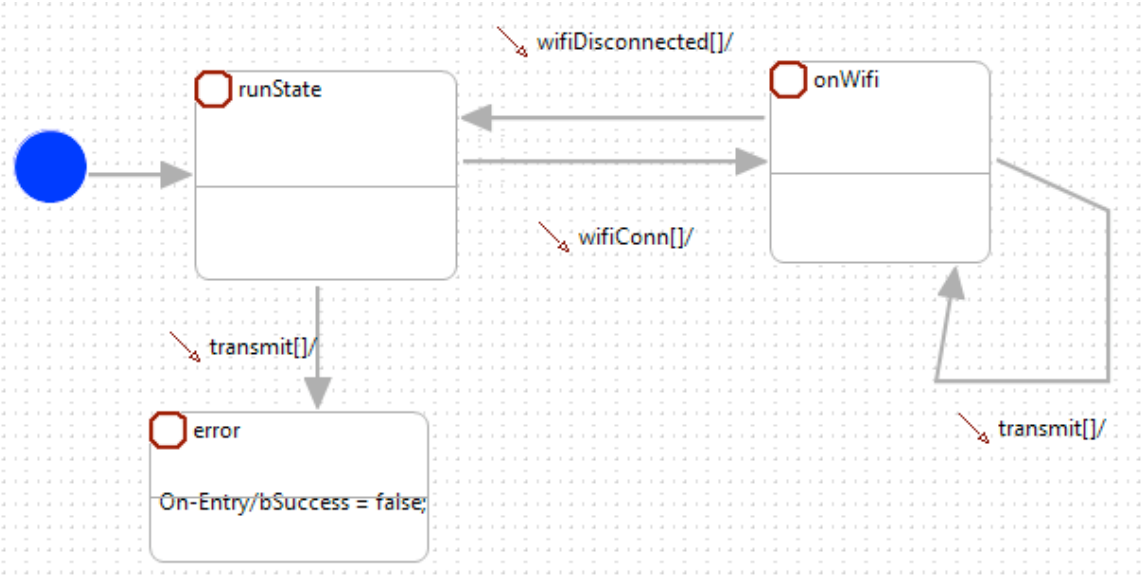

Figure 20. Statechart assertion for WiFi state

This statechart assertion could easily include the requirement for one hour between transmissions, but we want to ensure that we have the best understanding for what caused a failure. We separate these two requirements so we know that if one fails and not the other, what requirement caused the failure.

This statechart assertion covers the requirement that a transmission cannot start when not connected to Wi-Fi, but does not cover what needs to be done when the Wi-Fi

connection is lost during a transmission. This requirement is implicitly part of other requirements, and it will be handled in other statechart assertions.

D. LOG TRANSMISSION

The application has a time limit of thirty seconds after which the transmission must stop and the user must be notified that the log file did not transmit. Although the user would not see this, there is a time limit of five seconds in which the user must be notified. Additional requirements given are: a minimum time between transmissions of one hour, and the application must only transmit the log file when connected to Wi-Fi. We have already generated the assertion that handles half of the Wi-Fi connectivity requirement, but we need to ensure that an active transmission stops if no longer on Wi-Fi. It would be possible to build a statechart assertion that could handle all of these, but the complexity would increase significantly. We created three separate assertions to handle each one individually.

The first requirement is that a limit of thirty seconds is to be placed on the amount of time to transmit a log file. This is done both to reduce power consumption and to ensure there is not a security flaw that allows data to be streamed off the device. Thirty seconds may be too long to adequately support either of these, but the time can be easily changed to shorter times if the requirement changed. Figure 21 shows the requirement in statechart assertion form. We decided to combine this requirement with the requirement that an active transmission must stop when Wi-Fi connectivity is lost. This was done because the conditions for success and failure are duplicated for both requirements. During an active transmission, if a transmission completes, Wi-Fi connectivity is lost, or the thirty-second timer ends, then transmission should stop. If a transmission complete event occurs afterward, then it can be assumed that a transmission occurred, or continued when it should not have.

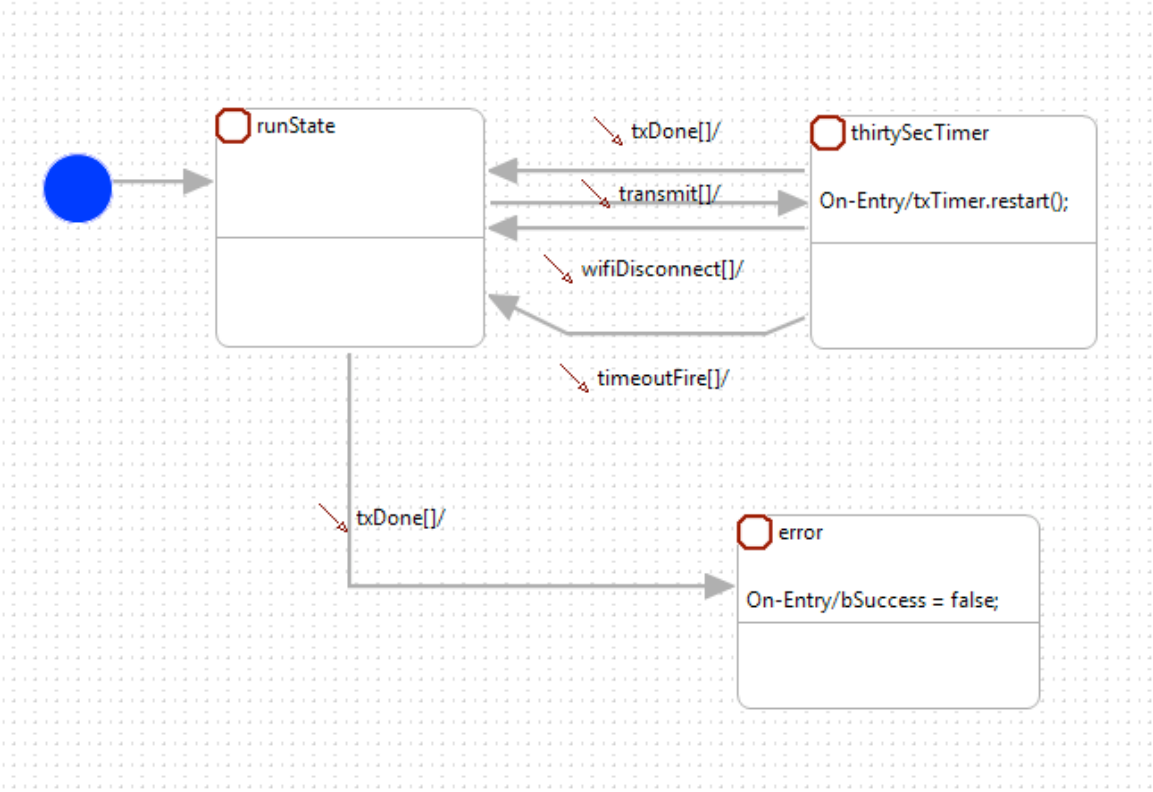

Figure 21. Statechart assertion limiting transmission time to 30 seconds

During the discussion regarding the requirements, we found that in order to save power and prevent logs containing only a few events from being uploaded, logs should only be uploaded when it has been an hour or more since the last log upload. As Figure 22 shows, after a ‘transmit’ event is received, it transitions to a new state, noMoreTransmit. At this point a timer is started, and once it expires, a transition back to the normal runState occurs. The timer is set to one hour. If at any point before the time expires, a new ‘transmit’ event occurs, then a transition to the error state occurs, indicating failure of the requirement.

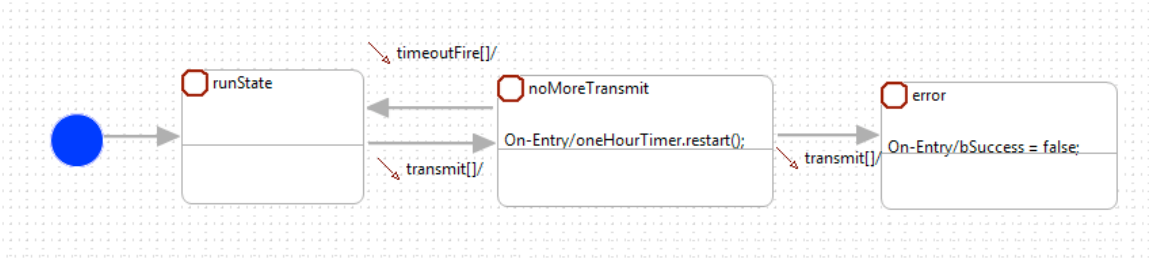

Figure 22. One-hour timer

There is no maximum time between uploads. If the requirement did exist, the resulting statechart assertion would look very similar to Figure 22. The two events, ‘timeoutFire’ and the ‘transmit’ from noMoreTransmit state to error state would be swapped. The assertion would then ensure that if a ‘transmit’ event did not occur within a specific time, then it would transition to the error state.

The final requirement needed is the user notification upon failure to transmit the log within the thirty-second window, shown in Figure 23. The first two states look very similar to Figure 22, but instead of the ‘timeoutFire’ causing a transition back to runState, it moves to a state that starts a five-second timer. If this timer expires without a ‘notifyUser’ event, then the requirement failed.

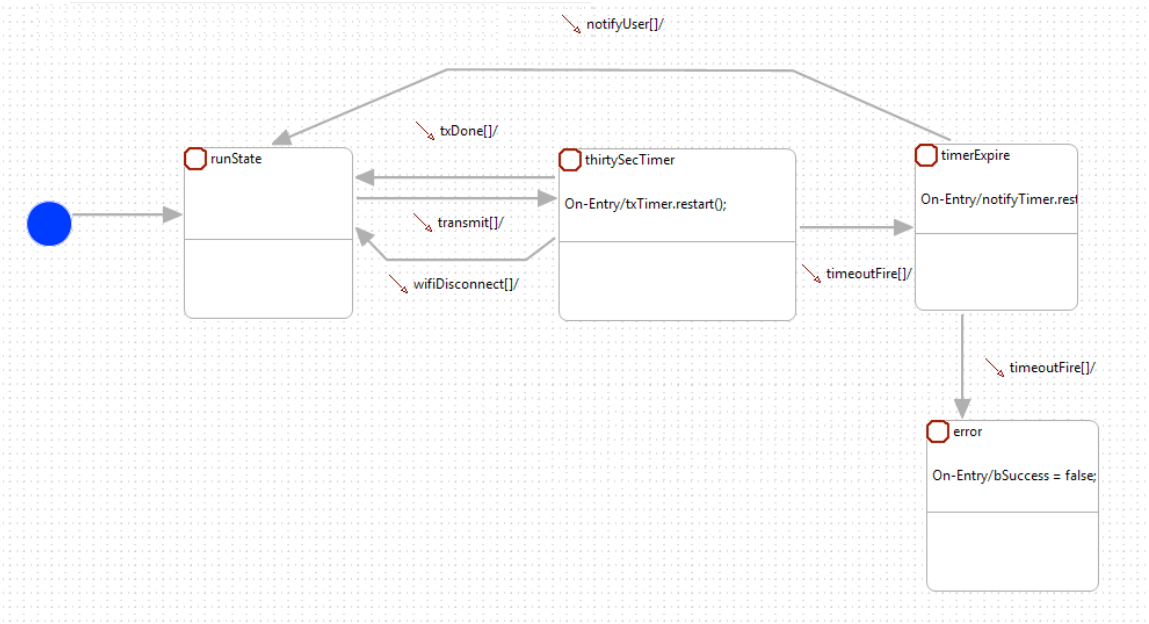

Figure 23. Five seconds to notify user of transmission failure

E. LOG PRE-PROCESSING

In our example, we use an application log file format that provides the name of an event and the time that it occurs. As stated before, the StateRover log file requires a unique format that our log file does not match. This will likely be the case often, for example, either the customer does not provide the exact format to the software developer, or a previously developed application is going to be evaluated against a different set of requirements. To convert the application log file into the format needed by StateRover, we created a Python program that will parse the log file and convert it into the correct format. The code written for our example is provided in Appendix A. After opening the file, the function parseData(inStr) takes each log entry and parses it into a tuple that hold the event type and the timestamp converted into an integer. The individual value of the integer is unimportant, but the difference between the integers for each timestamp represents the time difference between events.

Each tuple is then used to create a log entry in the StateRover log-file format. This is the function of the createLogArray(logArray) function. Using an if-then statement, the code looks for an event type and the prints the StateRover log format with the event name and the timestamp. As an example, the Wi-Fi Connection event in the application log looks like this:

WIFI_CONN @ 11/30/2012 01:07:44 PM
WIFI_DISCONNECT @ 11/30/2012 01:07:45 PM

After converting, the StateRover log entry looks like:

<event>
<sig><![CDATA[wifiConn]]></sig><time lang="c" unit="sec" val="1354309664" /></event>
<event>
<sig><![CDATA[wifiDisconn]]></sig><time lang="c" unit="sec" val="1354309665" /></event>

Note the difference between the two values, which represents the one-second difference between the events in the original log. The names of the events do not match because in our example, the event names in the application log do not match the statechart assertion event names in Figure 21 and Figure 23. We perform a one-to-one mapping of the names using the if-then statement. This mapping is not strictly necessary

because StateRover uses a namespace mapper to map the event name in the StateRover log to the event names in the statechart assertions. Using a mapping in the pre-processing state, though, can allow multiple events in the application log to be converted to the same event in the StateRover log, thus making the mapping from StateRover log event to the statechart assertion event simpler. For example, using two gpsUpdate events:

0 mps @ 11/30/2012 01:07:58 PM
1.08905 mps @ 11/30/2012 01:07:59 PM

Both events are GPS updates. Using the Python script, these are converted to:

<event>
<sig><![CDATA[gpsUpdate(float speed, int timestamp)]]></sig><args><arg type="float"name="speed" /><arg type="int" name="timeStamp" /></args><arg0 val="0.0" /><arg1 val="1354309678" />
<time lang="c" unit="sec" val="1354309678" /></event>
<event>
<sig><![CDATA[gpsUpdate(float speed, int timestamp)]]></sig><args><arg type="float" name="speed" /><arg type="int" name="timeStamp" /></args><arg0 val="1.08905" /><arg1 val="1354309679" />
<time lang="c" unit="sec" val="1354309679" /></event>

Without using a map and copying the event directly into the StateRover log file, we would get:

<event>
<sig><![CDATA[0 mps]]></sig>
<time lang="c" unit="sec" val="1354309678" /></event>
<event>
<sig><![CDATA[1.08905 mps]]></sig>
<time lang="c" unit="sec" val="1354309679" /></event>

When the StateRover log file is imported, it will read the two events as two different events, which would require a mapping from each event to the statechart assertion event, gpsUpdate. By using the script, both are converted into the same event and only require a single event to be mapped, in this case gpsUpdate(float speed, int timestamp). Once the file has been converted, we can now import the log into the project we created. We need the timestamp in two places in our StateRover log to provide the information to StateRover and the statechart assertion code. The line starting with <time lang…> is to provide a mechanism for StateRover to know how much to advance the timer in the statechart assertion. The reason we add the timestamp as an argument for

gpsUpdates is to allow the statechart assertions to ascertain the time difference between two non-consecutive gpsUpdates in order to calculate the average update per second.

F. IMPORTING AND EVALUATING A LOG

Prior to importing the newly created StateRover log, there are a couple of additional modifications we can do.

Double click on the file named AssertionTime.properties. It is located in the package named assertionrepository. There is only a single setting that defaults to “milli.” If not changed, will interpret the time change between two events as milliseconds. Since our log is in seconds, we need to make sure it is set to the value “sec.”

Once that has been added, we can add the StateRover log. Right click on the project, and select import. Since we converted the application log into the appropriate format, leave the radio button set to None. Find the log file to be imported and press Finish. Once done, two files will be added to the project. The log file as well as an XML file will be added to the project. The XML file is added to the LegalXmlLogs folder and will be the file StateRover will use. It should look the same as the imported log.

The final addition is to add the namespace map to the project. This must be done after importing the log file because it needs to reference the log file. To add the file, right click on the project and select New. Select Other, scroll down to the TimeRover folder, and then select Namespace Map. The map can be named to anything, but the default works fine. The field titled “New Source Log-File” needs to be set to the file created in the LegalXmlLogs folder during the import step. Our setup used:

C:/Users/author/Desktop/workspace/thesis/LegalXmlLogs/logOutput_legal.xml

The field titled New Target Assertion Repository needs to be set to the location of the project in which we are working. Ours looked like:

C:/Users/author/Desktop/workspace/thesis

Once entered, press the Finish button.

Once done, double click on the namespace map. If it is still the default, the file is named Map.namespace_map. An unmapped namespace map is shown in Figure 24.

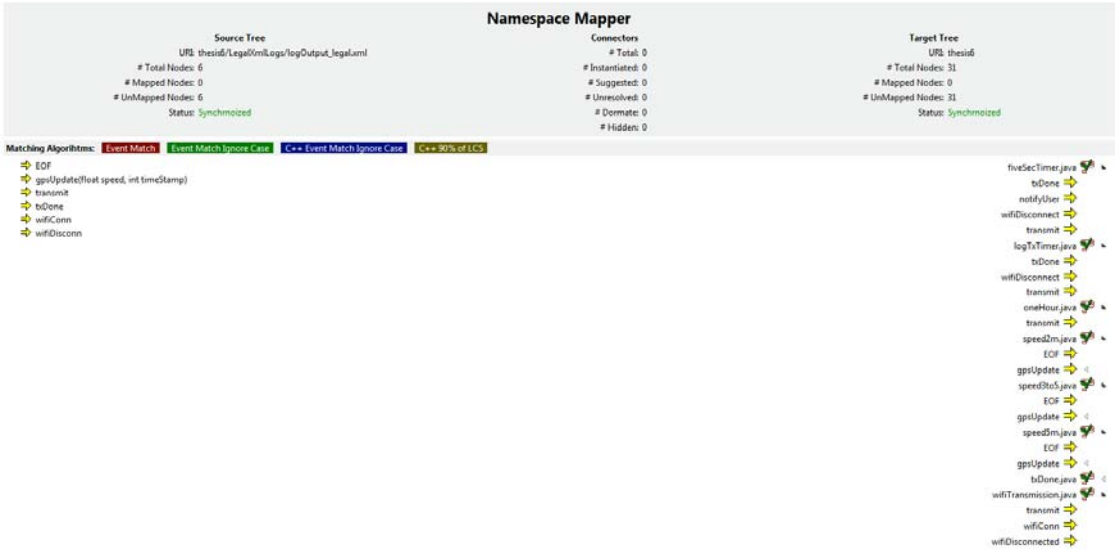

Figure 24. Unmapped events

On the left side are the events found in the StateRover log that was imported. On the right side are the events that exist in the imported statechart assertions. Right click on one of the events on the left and select Begin New Manual Connector. Select the event in a statechart assertion that it should be mapped to. This will be done for all events needed. During testing, targeted testing of one or more statechart assertions can be accomplished by mapping only the needed events. Figure 25 shows a a complete mapping of events.

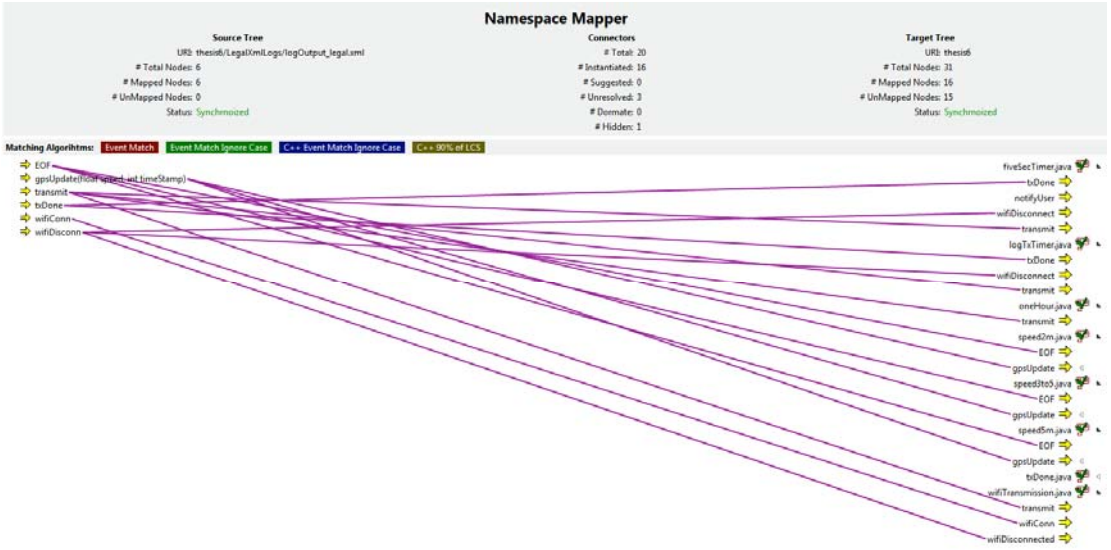

Figure 25. A complete map of events

Right clicking and select Instantiate All Connectors will turn the lines solid and complete the mapping. Once this is complete, the test can be run. Select Run button which looks like a green circle and white triangle in the tool bar. Figure 26 shows the desired result after testing one or more statechart assertions. If an assertion failure exists (i.e., a bSuccess variable in one of the assertions was set to false), the statechart assertion where it occurs will be listed on the left side under the header Statechart assertion failures.

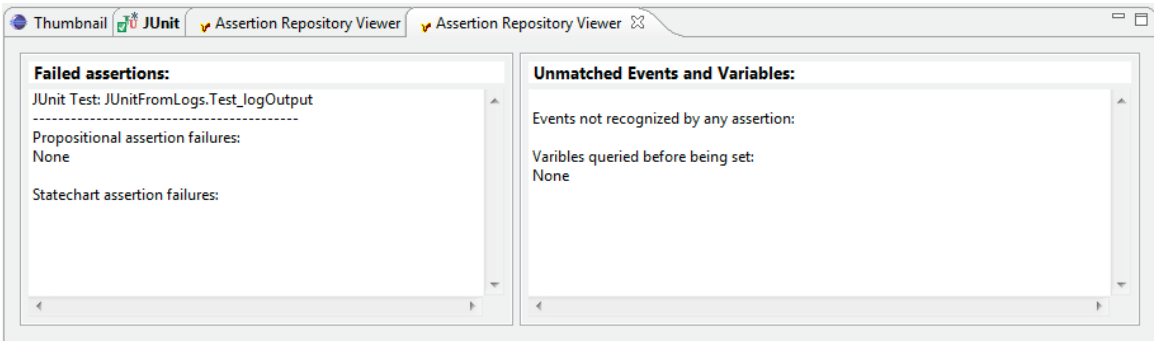

Figure 26. A test with zero failures

Further testing is much more simple now that everything is in place. When a new log file needs to be imported, delete the log file that was created when the first import occurred. The file in the LegalXMLFiles folder does not need to be deleted. Import the new file as before. The namespace map remains the same unless a new event occurred in the log file, or a statechart assertion had an event added. An issue experienced during writing is that occasionally changes made to statechart assertions do not work when the program is run subsequently. In those instances, it is suggested that project is cleaned so that the program will recompile from the source code. Do this by looking under the Project menu and clicking ‘Clean…’.

To validate the correct operation of the statechart assertions, we generated some log files containing errors. The log file in Figure 27 is an example snippet of what could be a typical log file.

```
1.0959 mps @ 11/30/2012 12:11:20 AM
1.3764 mps @ 11/30/2012 12:11:21 AM
0.9190 mps @ 11/30/2012 12:11:22 AM
0.7197 mps @ 11/30/2012 12:11:23 AM
WIFI_CONN @ 11/30/2012 12:11:23 AM
TX_START @ 11/30/2012 12:11:23 AM
1.9180 mps @ 11/30/2012 12:11:24 AM
0.5781 mps @ 11/30/2012 12:11:25 AM
0.3186 mps @ 11/30/2012 12:11:26 AM
0.7450 mps @ 11/30/2012 12:11:27 AM
0.1642 mps @ 11/30/2012 12:11:28 AM
0.7080 mps @ 11/30/2012 12:11:29 AM
1.7338 mps @ 11/30/2012 12:11:30 AM
WIFI_DISCONNECT @ 11/30/2012 12:11:30 AM
1.3015 mps @ 11/30/2012 12:11:31 AM
1.5235 mps @ 11/30/2012 12:11:32 AM
WIFI_CONN @ 11/30/2012 12:11:32 AM
0.8866 mps @ 11/30/2012 12:11:33 AM
1.4841 mps @ 11/30/2012 12:11:34 AM
TX_DONE @ 11/30/2012 12:11:34 AM
3.0644 mps @ 11/30/2012 12:11:35 AM
4.0769 mps @ 11/30/2012 12:11:35 AM
3.2224 mps @ 11/30/2012 12:11:36 AM
3.5195 mps @ 11/30/2012 12:11:36 AM
2.0872 mps @ 11/30/2012 12:11:37 AM
```

Figure 27. Sample log file

The events in the snippet show that the application that would have produced this would not have met the requirements. The log file completed transmission even after the device lost the Wi-Fi signal. This is easy to see in a short log file, but if this was a log file that was several kilobytes or megabytes in size, then analyzing the file would be tedious. Figure 28 shows the results of running the log file in StateRover. The log file failed the assertion statechart depicted in Figure 16 and Figure 22.

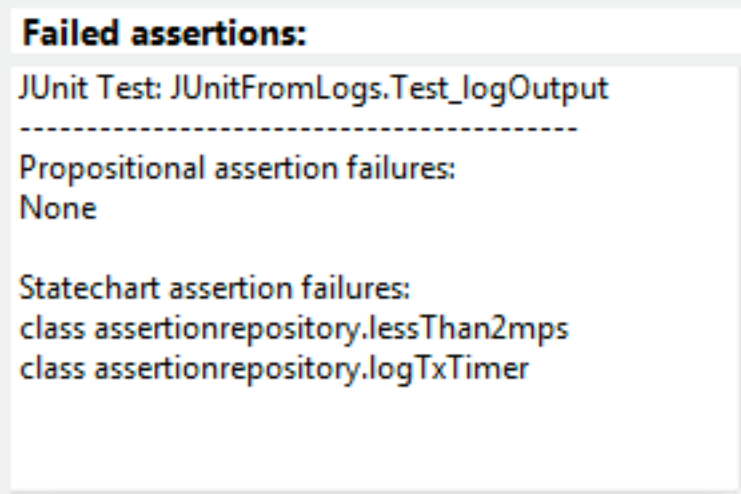

Figure 28. Failures after using the log file

G. LOG FILE ANALYSIS AND EVALUATION

Appendix C shows a log file produced by our GPS application. We used an Apple iPhone 4 running iOS 6 as the hardware platform to design and run the GPS application. The coding for the application was done in XCode 4.5.2. At the time of writing, the application was not complete and only produced GPS updates. By looking at the log file, it is easy to determine that this log file will not meet the requirements for the application. The most apparent is the inconsistency that there are too many GPS updates and they do not appear to vary based on the speed. We do not provide the StateRover log file of the application log because it is easy enough for the reader to perform the conversion with the code provided in Appendix A.

Once we import the log file, we produce a namespace map that looks like Figure 29. Since the only events in the StateRover log is are gpsUpdates, that is the only one that can be mapped. We map it to the gpsUpdate events in our statechart assertions.

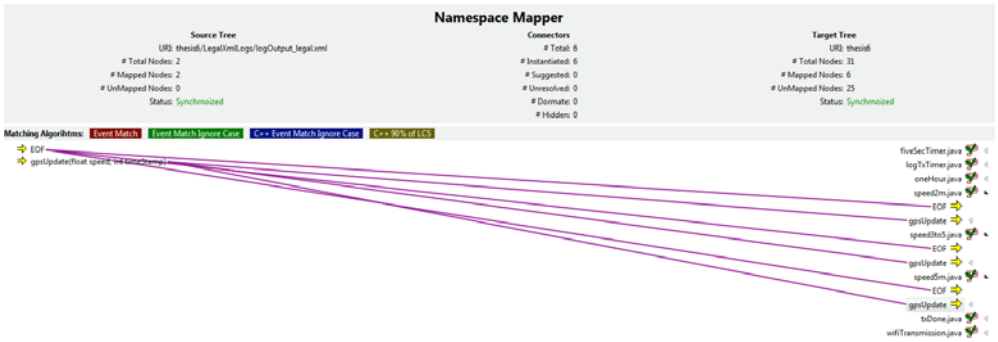

Figure 29. gpsUpdate-only mapping

The result when run is in Figure 30. All three assertions failed because the number of gpsUpdates received at some point in the log did not meet the requirement.

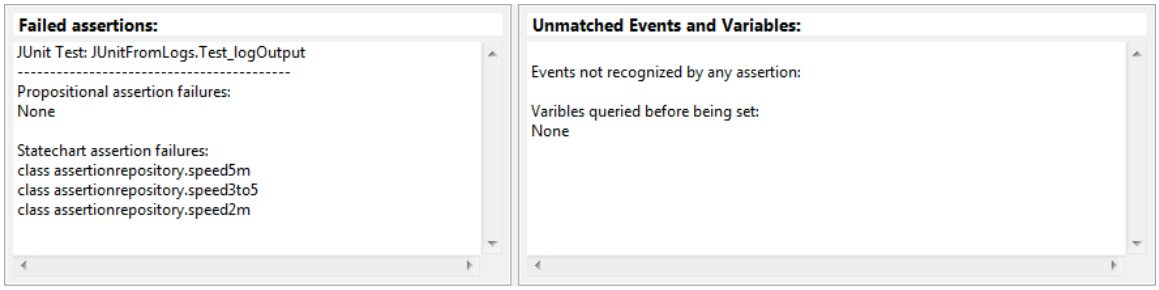

Figure 30. Failures reported

If this were a completed application, we could say with certainty that this application does not meet the requirements. As can be seen, the application failed three of the requirements. Based on this information, the customers could make a choice. They could alter the requirements so that this application would work; this would not usually be a first choice. If missions or lives depend on the fulfillment of the requirements, then that is not an option. The other option is that, if possible, the application would need to be changed, or a new application would need to be found.

H. APPLICATION CHANGE AND RESULTS

In our case, we can make adjustments to the application to attempt to meet the requirements. After making changes to how the GPS updates were handled, we were able to produce a log file as shown in Appendix D. On casual inspection, we cannot be sure if the file meets the requirements any better than the previous version. Once we have imported the log file into StateRover, we can quickly see that it does indeed meet the requirements.

From this case study, we can see the use of statechart assertions require a thorough understanding of the requirements, and a detailed development phase. Once developed though, many applications can be validated against the requirements with minimal changes.

V. LIMITATIONS, FUTURE WORK, AND CONCLUSION

Research into the use of statechart assertions has revealed some limitations into the technique, as well as some additional areas that are open to further research.

A. SUMMARY

This thesis presented a method for performing V&V on a DoD-centric mobile application using StateRover. The environment the DoD frequently operates in is abnormal to say the least, and is tough to emulate when attempting to perform V&V in a lab environment. It is important that an application is evaluated in the environment it is expected to operate in, especially since the programmers are probably unfamiliar with that environment. Log files provide direct insight into the operation of the application, and when used in the expected environment, can ensure a thorough and valid set of V&V tests. Combining the use of application log files and statechart assertions used in StateRover allows testers to evaluate the behavior of an application as it pertains to its adherence to the stated requirements. Statechart assertions provide a mechanism to represent application requirements into an easy to follow diagram that will be used by StateRover to automatically produce JUnit tests to evaluate the application log files. The modeling of the requirements rather than the application allows for multiple applications to be evaluated against the requirements, simplifying the analysis. The case study provides a non-trivial example of how the use of log files and statechart assertions provides a significant improvement in the V&V process of applications.

B. LESSONS LEARNED

For a statechart assertion to be effective when used to perform V&V, a tester must view the assertions from an observer's perspective. When developing statechart assertions, as computer scientists we want to design them like the software they represent. It makes sense that since a statechart assertion is supposed to test the operation of software then the assertion should operate like one. This led the authors down several

ill-conceived statechart assertions in which we tried to use the states and transitions as merely ways to run code for timers and variable reassignment. These original assertions did properly test that the software operated the way it was designed to operate, but of course this is axiomatic since software will always operate the way it is designed. One example is the use of variables extensively to keep track of the current state. In most programming languages, states are not available and variables are used to keep track of values (i.e., state of the system). For example, the first statechart assertions would use a variable to track if the software were connected to Wi-Fi or not. This would necessitate multiple transitions that would change the value of the variable dynamically which greatly increased the complexity of transitions in a statechart assertion. Besides the need for extra transitions, the states themselves would become virtually useless, as they were no longer used to represent the state of the system. In a true statechart, a state represents the system at a given moment. Thus, the system being in one state or another will indicate if Wi-Fi is connected or disconnected making a variable is unnecessary.

Another example of this incorrect thinking was the use of timers, for example in the one-hour timer requirement. These timers were included to model the various timer requirements. We essentially were representing the characteristics we were interested in, but correct operation of the timer in the statechart meant nothing since it is not linked to the operation of the software. What we needed to determine was if characteristics we were interested occurred correctly with respect to the stated time requirements. Timer in a statechart can be used to control the flow from one state to another to indicate that after the transition, the state of the system has changed. For the one-hour timer example, before the timer, the state meant that another log transmission was an error. The expiration of the timer would cause a transition to another state where another log transmission was ok. To accomplish this, we needed to examine the event that would start the one-hour time and the event/s that would signal when it was violated or not violated. This change in perspective does create some initial confusion, but must be overcome in order to create correct statechart assertions.

As mentioned in the case study, properly scoping a statechart assertion is important. While it is possible to create a single assertion that covers all the requirements, this is undesirable because of complexity, and development time. The statechart assertion developer must realize that individual requirements that are un-related should be separated when developing assertions. Understanding that the StateRover allowed multiple statechart assertions to run concurrently but separate can help.

C. LIMITATIONS

The effectiveness of statechart assertions to represent software behavior is still limited by the inherent inability of humans to articulate perfect requirement for the desired behavior. If a requirement is not provided, either because of the inability to define the requirement or the customer's need for the requirement is not known, then obviously it cannot be used to generate the statechart assertion that would evaluate the requirement. Articulating requirements and ensuring correctness is still a difficult problem that must be overcome in order to properly develop or evaluate a project.

Correctness of the statechart assertions depends on if the assertion correctly models the requirements. While using this technique is sound, if the assertions developed are incorrect, they will not properly evaluate the behavior of the software. The ability to exercise the statechart assertions against user-defined scenarios expressed as JUnit test cases helps validate the correctness of the assertions early in the requirements development process and reduce the risk of requirements errors.

Evaluating software to ensure that it does not contain code that is extraneous when compared to the software requirements is also not discussed in this thesis when evaluating the behavior of software. This is also a difficult problem that requires significant effort to ensure that the code used to operate the software does not contain code that is unnecessary and unwanted. The ability to specify undesirable behavior as statechart assertions helps in verifying the absence of some of these unnecessary codes.

D. FUTURE WORK

To further provide evidence that statechart assertion V&V is a unique and useful method, a complete beginning to end development cycle for a customer would be beneficial. We were unable to accomplish this due to time constraints. A follow-on software development example would provide that assurance that this method is indeed a step above other methods.

The statechart assertions used case study within the thesis was absolute yes or no. If an error occurred, then the requirement failed. It is conceivable that the assertions could be written to provide a number that could be used to produce a rating that would allow for comparing applications. This method would be more useful in a DoD environment when a customer is deciding between multiple applications.

A potential feature of statechart assertions for V&V is the possibility of evaluating more than one application that do not interact but their behavior with respect to the other may be of importance. A simple example could be an application that conducts texting and one that handles GPS such as the existing applications on the iPhone. If a requirement existed that the texting application could not text and drive, a statechart assertion could be written to ensure that at no time was the mobile device moving at driving speeds and have sent a text message. This is one advantage that statechart assertions have over any of the other V&V methods discussed previously.

APPENDIX A

```
import fileinput
import time
import datetime
import sys

numLogEntries = 0        # track how many log entries have
been processed so far.

#class to handle creating my own exception.
# Used in calcTimeInterval to return error notifying of
improper date/time format of log entry.
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

#Generate beginning of output file.
def prepOutput(outFile):
    outFile.write("<newtest>")

#Parse the log file entry into the format:
# [Timestamp, Log Entry]
def parseData(inStr):
    result = []
    ptrA = 0
    ptrB = 0
    ptrA = inStr.find('@')
    ptrB = inStr.find('M')
    temp = inStr[ptrA+2:ptrB-2].split(" ")

    if inStr.find("PM") != -1:
        dateString = "PM"
    else:
        dateString = "AM"

    #Stores in result an integer that represents the time
of the log entry.
    # Convert the timestamp into a time format, and then
convert into a floating point number, then recast as an
int.
    result.append(int(time.mktime(time.strptime(temp[0] +
temp[1] + dateString, "%m/%d/%Y%I:%M:%S%p"))))
```

```
    #Append everything before the @ after the timestamp.
The is the part tha identifies the type of log entry.
    result.append((inStr[:ptrA]).strip())

    return result


#Receives the log after it has been parsed and placed into
an array. Generates the appropriate state-rover log entry.
def createLogArray(logArray):

    global numLogEntries
    logEntryFunction = ""
    numLogEntries = numLogEntries + 1

    #gpsUpdate:
    # The first argument is the value of the speed.
    # The second argument is the timestamp. It is needed
    #  to determine different in time between two gpsUpdates
    if 'mps' in logArray[1]:
    logEntryFunction = "gpsUpdate(float speed, int
     timestamp)"
     speedNum = logArray[1].find('mps')

     outFile.write("\n<event>\n<sig><![CDATA["+
     logEntryFunction + "]]></sig><args><arg type=\"float\"
     name=\"speed\" /><arg type=\"int\" name=\"timeStamp\"
     /></args><arg0 val=\"" +
     str(float(logArray[1][:speedNum-1])) + "\" /><arg1
     val=\"" + str(logArray[0]) + "\" />")

     outFile.write("\n<time lang=\"c\" unit=\"sec\"
     val=\"" + str(logArray[0]) + "\" /></event>" )

     return;

    elif 'WIFI_CONN' in logArray[1]:
        #print "wifi connected"
        logEntryFunction = "wifiConn"

    elif 'WIFI_DISCONNECT' in logArray[1]:
        #print "wifi disconnected"
        logEntryFunction = "wifiDisconn"

    elif 'TX_START' in logArray[1]:
        #print "transmission started"
```

```
            logEntryFunction = “transmit”

        elif ‘DONE’ in logArray[1]:
            #print “transmission completed”
            logEntryFunction = “txDone”

        elif ‘USER’ in logArray[1]:
            #print “user notified”
            logEntryFunction = “userNotified”

        else:
            #print “Invalid Log Entry”
            raise MyError(“Invalid Log Entry”)

        #print out log entry into correct format
        outFile.write(“\n<event>\n<sig><![CDATA[“ +
        logEntryFunction + “]]></sig><time lang=\”c\”
        unit=\”sec\” val=\”“ + str(logArray[0]) + “\”
        /></event>“)

        return

if __name__ == “__main__”:

        outFile =
open(“/Users/cbonine/Documents/thesis/thesis/logOutput.log,
” ‘w’, 0)

        prepOutput(outFile)

        inFile =
open(“/Users/cbonine/Documents/thesis/thesis/inputFiles/log
5.gps,” ‘r’)
        for in_line in inFile.readlines():

            if len(in_line) == 2:
                continue
            try:
                createLogArray(parseData(in_line))
            except MyError as m:
                sys.exit(m)

        outFile.close()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

```
float prevSpeed = 0;
float newSpeed = 0;
int counter = 0;
int firstTime = 0;
int newTime = 0;
int curTime = 0;

//Function receives a float representing a speed of motion.
The number of seconds per update is returned as an int.
int speedCategory(float speed){
     if (speed <= 2) {return 0;}
     else if (speed > 5) {return 2;}
     else {return 1;}
}
//Function returns a boolean based on whether the new speed
and the previous speed are of the same update rate.
bool isSpeedChg(){
     return (speedCategory(prevSpeed) !=
speedCategory(newSpeed));
}
//Function returns a boolean based on whether the number of
seconds per update is within the expected range.
bool isCorrectAvg(){
     if (speedCategory(prevSpeed) == 0){
          if ((curTime/counter) >= updateRate) && (counter
>= 5))
               return true;
     }
     else if (speedCategory(prevSpeed) == 2)
          if ((curTime/counter) <= updateRate) && (counter
>= 5))
               return true;
     else {
          return (((curTime/counter) > 2 &&
(curTime/counter) <= 5) && counter >= 5);
     }
     return false;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C

-1 mps @ 01/03/2013 03:25:57 PM
-1 mps @ 01/03/2013 03:25:57 PM
-1 mps @ 01/03/2013 03:25:58 PM
0 mps @ 01/03/2013 03:26:55 PM
0 mps @ 01/03/2013 03:27:04 PM
0 mps @ 01/03/2013 03:27:10 PM
2.43334 mps @ 01/03/2013 03:27:11 PM
0 mps @ 01/03/2013 03:27:14 PM
1.39702 mps @ 01/03/2013 03:27:17 PM
1.23813 mps @ 01/03/2013 03:27:22 PM
1.05395 mps @ 01/03/2013 03:27:28 PM
1.13987 mps @ 01/03/2013 03:27:33 PM
1.00713 mps @ 01/03/2013 03:28:47 PM
1.85206 mps @ 01/03/2013 03:28:49 PM
2.52009 mps @ 01/03/2013 03:28:51 PM
2.83347 mps @ 01/03/2013 03:28:53 PM
3.19097 mps @ 01/03/2013 03:28:57 PM
3.48548 mps @ 01/03/2013 03:28:59 PM
3.43531 mps @ 01/03/2013 03:29:02 PM
0.75155 mps @ 01/03/2013 03:29:10 PM
1.71552 mps @ 01/03/2013 03:29:12 PM
2.83577 mps @ 01/03/2013 03:29:14 PM
7.72649 mps @ 01/03/2013 03:29:16 PM
7.37369 mps @ 01/03/2013 03:29:18 PM
7.69601 mps @ 01/03/2013 03:29:20 PM
7.7646 mps @ 01/03/2013 03:29:22 PM
7.9414 mps @ 01/03/2013 03:29:24 PM
7.73385 mps @ 01/03/2013 03:29:26 PM
7.7167 mps @ 01/03/2013 03:29:28 PM
6.23559 mps @ 01/03/2013 03:29:30 PM
7.5814 mps @ 01/03/2013 03:29:32 PM
7.09304 mps @ 01/03/2013 03:29:34 PM
7.52925 mps @ 01/03/2013 03:29:36 PM
7.66508 mps @ 01/03/2013 03:29:38 PM
6.58963 mps @ 01/03/2013 03:29:40 PM
7.95306 mps @ 01/03/2013 03:29:42 PM
7.94375 mps @ 01/03/2013 03:29:44 PM
8.46784 mps @ 01/03/2013 03:29:46 PM
7.87807 mps @ 01/03/2013 03:29:48 PM
8.51375 mps @ 01/03/2013 03:29:50 PM
7.97661 mps @ 01/03/2013 03:29:52 PM
6.30209 mps @ 01/03/2013 03:29:54 PM
7.71578 mps @ 01/03/2013 03:29:56 PM

```
7.47224 mps @ 01/03/2013 03:29:58 PM
8.54379 mps @ 01/03/2013 03:30:00 PM
7.85834 mps @ 01/03/2013 03:30:02 PM
7.04557 mps @ 01/03/2013 03:30:04 PM
6.71061 mps @ 01/03/2013 03:30:06 PM
7.67473 mps @ 01/03/2013 03:30:08 PM
7.5352 mps @ 01/03/2013 03:30:10 PM
8.26685 mps @ 01/03/2013 03:30:12 PM
7.42276 mps @ 01/03/2013 03:30:14 PM
7.43271 mps @ 01/03/2013 03:30:16 PM
6.994 mps @ 01/03/2013 03:30:18 PM
7.05431 mps @ 01/03/2013 03:30:20 PM
8.32787 mps @ 01/03/2013 03:30:22 PM
7.69903 mps @ 01/03/2013 03:30:24 PM
…
17.5019 mps @ 01/03/2013 03:38:35 PM
17.39016 mps @ 01/03/2013 03:38:36 PM
17.45489 mps @ 01/03/2013 03:38:37 PM
17.54906 mps @ 01/03/2013 03:38:38 PM
18.24514 mps @ 01/03/2013 03:38:39 PM
18.75679 mps @ 01/03/2013 03:38:40 PM
18.49855 mps @ 01/03/2013 03:38:41 PM
18.67274 mps @ 01/03/2013 03:38:42 PM
18.53703 mps @ 01/03/2013 03:38:43 PM
18.4852 mps @ 01/03/2013 03:38:44 PM
18.06501 mps @ 01/03/2013 03:38:45 PM
18.13891 mps @ 01/03/2013 03:38:46 PM
18.22478 mps @ 01/03/2013 03:38:47 PM
18.27673 mps @ 01/03/2013 03:38:48 PM
18.04896 mps @ 01/03/2013 03:38:49 PM
17.96455 mps @ 01/03/2013 03:38:50 PM
17.82744 mps @ 01/03/2013 03:38:51 PM
17.74577 mps @ 01/03/2013 03:38:52 PM
17.79791 mps @ 01/03/2013 03:38:53 PM
18.04387 mps @ 01/03/2013 03:38:54 PM
18.16834 mps @ 01/03/2013 03:38:55 PM
18.03198 mps @ 01/03/2013 03:38:56 PM
17.76463 mps @ 01/03/2013 03:38:57 PM
17.62269 mps @ 01/03/2013 03:38:58 PM
17.08219 mps @ 01/03/2013 03:38:59 PM
16.66813 mps @ 01/03/2013 03:39:00 PM
16.92726 mps @ 01/03/2013 03:39:01 PM
16.58836 mps @ 01/03/2013 03:39:02 PM
16.18446 mps @ 01/03/2013 03:39:03 PM
16.307 mps @ 01/03/2013 03:39:04 PM
16.307 mps @ 01/03/2013 03:39:05 PM
```

15.5542 mps @ 01/03/2013 03:39:06 PM
14.4142 mps @ 01/03/2013 03:39:07 PM
13.91462 mps @ 01/03/2013 03:39:08 PM
13.15717 mps @ 01/03/2013 03:39:09 PM
11.03492 mps @ 01/03/2013 03:39:10 PM
8.59739 mps @ 01/03/2013 03:39:11 PM
5.51583 mps @ 01/03/2013 03:39:13 PM
1.59482 mps @ 01/03/2013 03:39:18 PM
2.29419 mps @ 01/03/2013 03:39:23 PM
4.65751 mps @ 01/03/2013 03:39:28 PM
6.60028 mps @ 01/03/2013 03:39:30 PM
9.67114 mps @ 01/03/2013 03:39:32 PM
11.12432 mps @ 01/03/2013 03:39:33 PM
11.87481 mps @ 01/03/2013 03:39:34 PM
12.1719 mps @ 01/03/2013 03:39:35 PM
11.60195 mps @ 01/03/2013 03:39:36 PM
9.90325 mps @ 01/03/2013 03:39:37 PM
4.65094 mps @ 01/03/2013 03:39:39 PM
5.892 mps @ 01/03/2013 03:39:41 PM
6.66678 mps @ 01/03/2013 03:39:43 PM
6.52942 mps @ 01/03/2013 03:39:45 PM
6.14266 mps @ 01/03/2013 03:39:47 PM
5.26809 mps @ 01/03/2013 03:39:49 PM
5.01334 mps @ 01/03/2013 03:39:52 PM
6.4921 mps @ 01/03/2013 03:39:54 PM
7.41495 mps @ 01/03/2013 03:39:56 PM
7.54477 mps @ 01/03/2013 03:39:58 PM
7.7361 mps @ 01/03/2013 03:40:00 PM
7.90536 mps @ 01/03/2013 03:40:02 PM
8.19249 mps @ 01/03/2013 03:40:04 PM
8.53728 mps @ 01/03/2013 03:40:06 PM
8.14979 mps @ 01/03/2013 03:40:08 PM
8.22218 mps @ 01/03/2013 03:40:10 PM
7.4163 mps @ 01/03/2013 03:40:12 PM
7.23022 mps @ 01/03/2013 03:40:14 PM
6.72486 mps @ 01/03/2013 03:40:16 PM
6.10299 mps @ 01/03/2013 03:40:18 PM
5.82705 mps @ 01/03/2013 03:40:20 PM
6.02752 mps @ 01/03/2013 03:40:22 PM
5.83089 mps @ 01/03/2013 03:40:24 PM
5.49173 mps @ 01/03/2013 03:40:26 PM
4.00314 mps @ 01/03/2013 03:40:28 PM
1.09299 mps @ 01/03/2013 03:40:42 PM

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D

-1 mps @ 01/03/2013 04:49:27 PM
-1 mps @ 01/03/2013 04:50:24 PM
-1 mps @ 01/03/2013 04:50:24 PM
-1 mps @ 01/03/2013 04:50:27 PM
-1 mps @ 01/03/2013 04:50:47 PM
0 mps @ 01/03/2013 04:50:51 PM
1.75552 mps @ 01/03/2013 04:51:03 PM
5.24983 mps @ 01/03/2013 04:51:14 PM
5.08223 mps @ 01/03/2013 04:51:16 PM
4.91206 mps @ 01/03/2013 04:51:19 PM
4.83426 mps @ 01/03/2013 04:51:21 PM
4.09447 mps @ 01/03/2013 04:51:25 PM
4.05589 mps @ 01/03/2013 04:51:28 PM
4.28685 mps @ 01/03/2013 04:51:31 PM
4.50909 mps @ 01/03/2013 04:51:33 PM
4.84993 mps @ 01/03/2013 04:51:35 PM
5.261 mps @ 01/03/2013 04:51:37 PM
7.86725 mps @ 01/03/2013 04:51:39 PM
8.09164 mps @ 01/03/2013 04:51:41 PM
8.03185 mps @ 01/03/2013 04:51:43 PM
7.78858 mps @ 01/03/2013 04:51:45 PM
7.30402 mps @ 01/03/2013 04:51:47 PM
7.56517 mps @ 01/03/2013 04:51:49 PM
7.70443 mps @ 01/03/2013 04:51:51 PM
7.8417 mps @ 01/03/2013 04:51:53 PM
6.23378 mps @ 01/03/2013 04:51:55 PM
4.79682 mps @ 01/03/2013 04:51:57 PM
3.69489 mps @ 01/03/2013 04:52:00 PM
3.16958 mps @ 01/03/2013 04:52:03 PM
2.84119 mps @ 01/03/2013 04:52:06 PM
3.48196 mps @ 01/03/2013 04:52:10 PM
5.66709 mps @ 01/03/2013 04:52:13 PM
6.09716 mps @ 01/03/2013 04:52:15 PM
2.56607 mps @ 01/03/2013 04:52:18 PM
4.887 mps @ 01/03/2013 04:52:21 PM
8.46612 mps @ 01/03/2013 04:52:23 PM
10.43208 mps @ 01/03/2013 04:52:24 PM
11.30968 mps @ 01/03/2013 04:52:25 PM
10.90521 mps @ 01/03/2013 04:52:26 PM
11.06819 mps @ 01/03/2013 04:52:27 PM
10.57058 mps @ 01/03/2013 04:52:28 PM
9.25469 mps @ 01/03/2013 04:52:29 PM
5.09837 mps @ 01/03/2013 04:52:31 PM

```
4.72216 mps @ 01/03/2013 04:52:34 PM
7.66597 mps @ 01/03/2013 04:52:36 PM
11.51016 mps @ 01/03/2013 04:52:38 PM
12.97724 mps @ 01/03/2013 04:52:39 PM
13.92318 mps @ 01/03/2013 04:52:40 PM
14.74832 mps @ 01/03/2013 04:52:41 PM
15.52247 mps @ 01/03/2013 04:52:42 PM
15.84677 mps @ 01/03/2013 04:52:43 PM
15.85587 mps @ 01/03/2013 04:52:44 PM
15.96836 mps @ 01/03/2013 04:52:45 PM
16.30545 mps @ 01/03/2013 04:52:46 PM
16.54926 mps @ 01/03/2013 04:52:47 PM
16.45933 mps @ 01/03/2013 04:52:48 PM
16.63872 mps @ 01/03/2013 04:52:49 PM
16.5046 mps @ 01/03/2013 04:52:50 PM
16.97579 mps @ 01/03/2013 04:52:51 PM
16.92315 mps @ 01/03/2013 04:52:52 PM
16.93476 mps @ 01/03/2013 04:52:53 PM
16.96807 mps @ 01/03/2013 04:52:54 PM
16.94887 mps @ 01/03/2013 04:52:55 PM
17.00163 mps @ 01/03/2013 04:52:56 PM
17.25296 mps @ 01/03/2013 04:52:57 PM
17.39488 mps @ 01/03/2013 04:52:58 PM
17.35574 mps @ 01/03/2013 04:52:59 PM
17.30094 mps @ 01/03/2013 04:53:00 PM
…
10.49857 mps @ 01/03/2013 06:06:24 PM
10.62999 mps @ 01/03/2013 06:06:25 PM
10.13194 mps @ 01/03/2013 06:06:26 PM
9.53014 mps @ 01/03/2013 06:06:27 PM
9.50657 mps @ 01/03/2013 06:06:28 PM
9.99946 mps @ 01/03/2013 06:06:29 PM
8.00262 mps @ 01/03/2013 06:06:31 PM
6.51004 mps @ 01/03/2013 06:06:33 PM
3.49945 mps @ 01/03/2013 06:06:36 PM
5.21874 mps @ 01/03/2013 06:08:05 PM
7.66158 mps @ 01/03/2013 06:08:07 PM
9.28941 mps @ 01/03/2013 06:08:09 PM
10.39878 mps @ 01/03/2013 06:08:11 PM
11.46583 mps @ 01/03/2013 06:08:12 PM
11.90638 mps @ 01/03/2013 06:08:13 PM
13.24221 mps @ 01/03/2013 06:08:14 PM
13.59304 mps @ 01/03/2013 06:08:15 PM
14.08983 mps @ 01/03/2013 06:08:16 PM
14.1081 mps @ 01/03/2013 06:08:17 PM
15.26516 mps @ 01/03/2013 06:08:18 PM
```

14.90772 mps @ 01/03/2013 06:08:19 PM
14.70042 mps @ 01/03/2013 06:08:20 PM
15.039 mps @ 01/03/2013 06:08:21 PM
15.04602 mps @ 01/03/2013 06:08:22 PM
15.1848 mps @ 01/03/2013 06:08:23 PM
14.86588 mps @ 01/03/2013 06:08:24 PM
15.15426 mps @ 01/03/2013 06:08:25 PM
15.02676 mps @ 01/03/2013 06:08:26 PM
15.22184 mps @ 01/03/2013 06:08:27 PM
14.40649 mps @ 01/03/2013 06:08:28 PM
14.66752 mps @ 01/03/2013 06:08:29 PM
14.31351 mps @ 01/03/2013 06:08:30 PM
13.95425 mps @ 01/03/2013 06:08:31 PM
14.01046 mps @ 01/03/2013 06:08:32 PM
13.46309 mps @ 01/03/2013 06:08:33 PM
13.47141 mps @ 01/03/2013 06:08:34 PM
14.07406 mps @ 01/03/2013 06:08:35 PM
14.09203 mps @ 01/03/2013 06:08:36 PM
14.87431 mps @ 01/03/2013 06:08:37 PM
15.14994 mps @ 01/03/2013 06:08:38 PM
15.40922 mps @ 01/03/2013 06:08:39 PM
15.96022 mps @ 01/03/2013 06:08:40 PM
16.12285 mps @ 01/03/2013 06:08:41 PM
16.58607 mps @ 01/03/2013 06:08:42 PM
16.11967 mps @ 01/03/2013 06:08:43 PM
15.52753 mps @ 01/03/2013 06:08:44 PM
16.05248 mps @ 01/03/2013 06:08:45 PM
15.18414 mps @ 01/03/2013 06:08:46 PM
14.09942 mps @ 01/03/2013 06:08:47 PM
12.28017 mps @ 01/03/2013 06:08:48 PM
11.41943 mps @ 01/03/2013 06:08:49 PM
10.39643 mps @ 01/03/2013 06:08:50 PM
7.84293 mps @ 01/03/2013 06:08:52 PM
3.95623 mps @ 01/03/2013 06:08:55 PM
4.63885 mps @ 01/03/2013 06:08:57 PM
5.89433 mps @ 01/03/2013 06:08:59 PM
6.18385 mps @ 01/03/2013 06:09:01 PM
5.9428 mps @ 01/03/2013 06:09:03 PM
5.5657 mps @ 01/03/2013 06:09:05 PM
5.20912 mps @ 01/03/2013 06:09:07 PM
3.84883 mps @ 01/03/2013 06:09:10 PM
3.11701 mps @ 01/03/2013 06:09:14 PM
2.52596 mps @ 01/03/2013 06:09:18 PM
0.00867 mps @ 01/03/2013 06:09:37 PM
-1 mps @ 01/03/2013 06:09:52 PM
-1 mps @ 01/03/2013 06:10:06 PM

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Alic, J. A., Branscomb, Lewis M., & Brooks, Harvey (1992). *Beyond spinoff: Military and commercial technologies in a changing world.* Boston: Harvard Business Press.

Bergue Alves, M. C., Drusinsky, D., Michael, J. B., & Shing, M.-T. (2011). Formal validation and verification of space flight software using statechart-assertion and runtime execution monitoring. In *Proceedings of the 6th International Conference on System of Systems Engineering* (pp. 155–160). doi: 10.1109/SYSOSE.2011.5966590

B'Far, R. (2004). Mobile computing principles: designing and developing mobile applications with UML and XML. Cambridge, England: Cambridge University Press.

Bo, J., Xiang, L., & Xiaopeng, G. (2007). Mobile Test: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings of the Second International Workshop on Automation of Software Test.* (pp. 8). doi: 10.1109/AST.2007.9

BSQUARE. (2013). *TestQuest automated testing*. Retrieved from BSQUARE website: http://www.bsquare.com/products/testquest-automated-testing-platform

Carey, N. (2005). *Establishing pedestrian walking speeds (Draft).* Portland, OR: Portland State University. Retrieved from http://www.westernite.org/datacollectionfund/2005/psu_ped_summary.pdf

Clarke, E. M., & Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys, 28*, 626–643. doi: 10.1.1.47.2696

Darcey, L., & Condor, S. (2009, August 10). *Introducing Android: A brief history of mobile application development.* Retrieved from http://www.informit.com/articles/article.aspx?p=1388959

Delamaro, M. E., Vincenzi, A. M., & Maldonado, J. C. (2006). A strategy to perform coverage testing of mobile applications. In *Proceedings of the International Workshop on Automation of Software Test* (pp. 118–124). doi: 10.1145/1138929.1138952

Drusinsky, D., Michael, J. B., & Shing, M.-T. (2007). *The three dimensions of formal validation and verification of reactive system behaviors (*NPS-CS-07–008). Monterey: Naval Postgraduate School, Department of Computer Science.

Drusinsky, D., Michael, J. B., Otani, T. W., & Shing, M.-T. (2008). Validating UML statechart-based assertions libraries for improved reliability and assurance. In *Proceedings of the The Second International Conference on Secure System Integration and Reliability Improvement* (pp. 47–51). doi: 10.1109/SSIRI.2008.54

Ellman, J., Livergood, R., Morrow, D., & Sanders, G. (2011, May). *Center for stragetic and international studies.* Retrieved from U.S. Department of Defense Contract Spending and the Supporting Industrial Base: http://csis.org/files/publication/110506_CSIS_Defense_Contract_Trends-sm2.pdf

Ernst, M. D. (2003). Static and dynamic analysis: synergy and duality. In *Proceedings of the International Conference on Software Engineering Workshop on Dynamic Analysis 2003* (pp. 25–28*).* Portland, OR. Retrieved from http://homes.cs.washington.edu/~mernst/pubs/woda2003-proceedings.pdf

Grey, J. (2006, July 7). Private military contractors: A short history. Retrieved from http://polosbastards.com/pb/private-military-contractors-pmcs-a-short-history/

Harel, D. (1987). Statecharts: A visual formalism for complex systems. The Weizmann Institute of Science, Department of Applied Mathematics. Holland: Elsevier Schience Publishers. doi: 10.1.1.20.4799

Juristo, N., & Vegas, S. (2003). Functional testing, structural testing and code reading: What fault types do they each detect? In R. Conradi & A. I. Wang (Eds.) *Empirical Methods and Studies in Software Engineering* (pp. 208–232). Madrid: Universidad Politécnica de Madrid.

Kenyon, H. (2012, January 27). DISA office to manage mobile devices, online app store. *Government Computer News.* Retrieved June 22, 2012, from http://gcn.com/articles/2012/01/27/disa-launches-program-office-to-manage-mobile-devices.aspx

Michael, J. B., Drusinsky, D., Otani, T. W., & Shing, M.-T. (2011). Verification and Validation for Trustworthy Software Systems. *IEEE Software, 28* (4*),* 86–92. doi: 10.1109/MS.2011.151

monkeyrunner. (n.d.). *monkeyrunner*. Retrieved January 23, 2013, from android developers website: http://developer.android.com/tools/help/monkeyrunner_concepts.html

Muccini, H., Francesco, A. D., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. *In the Proceedings of the 7th International Workshop on Automation of Software Test* (pp. 29–35). doi: 10.4204/EPTCS.61

Myers, G. J. (2004). The art of software testing (Second Edition ed.). Hoboken, New Jersey, USA: John Wiley & Sons, Inc.

Office of Management and Budget (OMB). (n.d.). Table 9.8 Composition of outlays for the conduct of research and development: 1949–2013. Retrieved from OMB Historical Tables: http://www.whitehouse.gov/omb/budget/Historicals

Randall, Jr., D. A., & Seaberry, C. M. (2009). *Contingency contracting and the IT manager: Today's challenges and future implications* (Master's thesis, Naval Postgraduate School). Retrieved from http://edocs.nps.edu/npspubs/scholarly/theses/2009/Mar/09Mar_Randall.pdf

Robotium. (n.d.). *User scenario testing for Android*. Retrieved from Robotium: http://code.google.com/p/robotium/

Sagar, I. (2012, June 29). Before iPhone and Android Came Simon, the First Smartphone. *Bloomberg Businessweek Technology*. Retrieved from http://www.businessweek.com/articles/2012–06–29/before-iphone-and-android-came-simon-the-first-smartphone

Stowsky, J. (2005). From spin-off to spin-on: Redefining the military's role in technology development. *Berkeley Roundtable on the International Economy.* Berkeley: UC Berkeley. Retrieved from http://www.escholarship.org/uc/item/0tf8v3c7#page-1

Department of Defense (DoD). (2012). *Department of Defense mobile device strategy.* Retrieved from http://www.defense.gov/news/dodmobilitystrategy.pdf

The evolution of cell phone design between 1983–2009. (2009, May 22). Retrieved January 17, 2013, from WebDesigner Depot website: http://www.webdesignerdepot.com/2009/05/the-evolution-of-cell-phone-design-between-1983-2009/

Wallace, D. R., & Fujii, R. U. (1989). Software verification and validation: An overview. *IEEE Software* , pp. 10–17. doi: 10.1109/52.28119

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Thomas Otani
   Naval Postgraduate School
   Monterey, California

4. Dr. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

5. Dr. Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

6. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, California

7. Mr. John Gibson
   Naval Postgraduate School
   Monterey, California

8. Dr. Keith Snider
   Naval Postgraduate School
   Monterey, California

9. Ms. Karey L. Shaffer
   Naval Postgraduate School
   Monterey, California

10. Ms. Tera Yoder
    Naval Postgraduate School
    Monterey, California

11. Mr. Tao Rocha
    SPAWAR Atlantic
    Charleston, South Carolina

12. CDR. Kurt Rothenhaus
PEO C4I, PMW/A 170
San Diego, California

13. Mr. Richard Delgado Jr.
Joint Interoperability Test Command
Fort Huachuca, Arizona

14. Mr. Randon Herrin
Joint Interoperability Test Command
Fort Huachuca, Arizona